



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

University of Alberta

A Template-Based Approach to Structuring Distributed
Algorithms Using a Network of Workstations

by



Ajit Singh

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta
Spring, 1992



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73023-4

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Ajit Singh

TITLE OF THESIS: A Template-Based Approach to Structuring Distributed
Algorithms Using a Network of Workstations

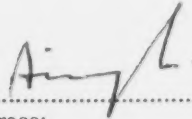
DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)



Permanent Address:

c/o Dr. A. D. Taskar

F-Block, 5B Saket,

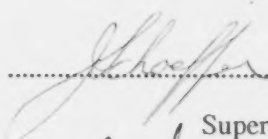
New Delhi 110 017, INDIA.

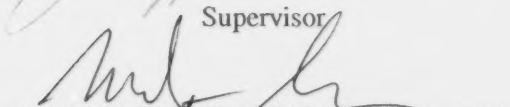
Dated October 21, 1991

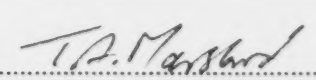
THE UNIVERSITY OF ALBERTA

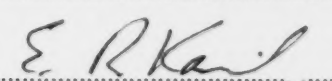
FACULTY OF GRADUATE STUDIES AND RESEARCH

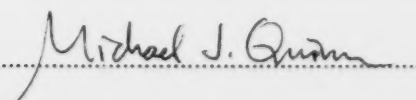
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations** submitted by **Ajit Singh** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.


.....
Supervisor


.....
Supervisor

.....

.....


.....


.....

Date October 21, 1991

To my wife Ravi, and children Charu and Aditi

Abstract

Despite rapid growth in workstation and networking technologies, shared processing in a workstation environment continues to face challenging problems because of factors such as high communication costs, wide fluctuations in the available computing power, ownership constraints, and dissimilar processor capabilities. This thesis presents a computational model and system, called *FrameWorks*, for the generation of parallel or distributed applications in such an environment. The model is based on a novel concept known as *template attachment*, which allows separation of the specification of the structure of a distributed application (communication, synchronization, scheduling, and processor binding) from the program code of the application. A distributed application in *FrameWorks* is represented by a directed graph. Except for blocking and non-blocking calls to other nodes, a node consists entirely of sequential procedures. All other synchronization/communication characteristics, as well as the processor allocation constraints of a node are described via specially designed graphical icons called *templates*, and an interactive graphical interface. A selection of templates is provided by the system. By attaching an appropriate set of templates to each node, most of the commonly occurring distributed process graphs can be rapidly constructed. A specific template of *FrameWorks*, called *contractors*, provides a novel method for distributing work in environments with varying amounts of available computing power. Contractors dynamically contract out work to employee processes and are adaptive to the changing constraints of the environment. To demonstrate the practical feasibility of the model, two versions of a visual system based on the *FrameWorks* model have been developed for the Unix based workstation environment. Our experience so far with the *FrameWorks* system indicates that the template-based approach is viable and can potentially offer significant ease and flexibility in developing and restructuring distributed applications using a network of processors.

Acknowledgements

This thesis has been possible because of the encouragement and support I received from Dr. Jonathan Schaeffer. The confidence that he showed in me made several difficult tasks much easier. Dr. Jonathan Schaeffer and Dr. Mark Green (my supervisors) helped me every step of the way and provided technical, financial and moral support. The quality and readability of the thesis is largely due to their effort in reading the earlier drafts of the thesis.

Thanks are due to Dr. Michael Quinn, Dr. Tony Marsland, and Dr. E. Kanasewich, my examination committee members, for the meticulous and thorough evaluation of the thesis and their comments on improving its quality. I also thank Dr. Xiaobo Li for several stimulating discussions and the financial support that he provided for sending me to attend a course on Parallel Computing at Argonne National Laboratory, Illinois.

Many thanks to the support staff members of the Department of Computing Science who provide such a friendly and efficient environment for research work. Steve Sutphen's technical excellence helped me on so many occasions. Edith Drummond is the graduate office person I could always count on. The department along with NSERC provided me with the much needed financial assistance via teaching and research assistantships.

Several persons helped me in implementing *FrameWorks*. I received noteworthy help from summer students Glenn Klettke and Wade Holst, and fellow graduate student Gurminder Singh. Yigong Wong agreed to use *FrameWorks* for his Ph.D. work and provided very useful comments. Paul Lu helped me finally submit this thesis to the Faculty of Graduate Studies and Research.

Most of all, my wife Ravi, my son Charu, and my daughter Aditi made it so natural and easy for me to complete my work. None complained about the numerous evenings, weekends, and holidays when I used to be missing from home. They patiently waited for "daddy's degree" to be completed.

Table of Contents

1.0 Chapter 1: Shared Processing in a Workstation Environment ...	1
1.1 The Growth of Workstation Environments	1
1.2 The problem	2
1.3 The FrameWorks Approach	3
1.4 Contribution of the Thesis	4
1.5 Scope of the Thesis	6
1.6 Organization of the Thesis	7
1.7 Conclusions	7
2.0 Chapter 2: Review of Basic Issues	9
2.1 Introduction	9
2.2 Scope Definition	10
2.3 Review of Basic Techniques for Parallelism	12
2.3.1 Specification of Parallel Execution	12
2.3.1.1 Comparative Analysis	12
2.3.2 Interaction among Parallel Processes	14
2.3.2.1 Process Interaction in a Shared Memory Environment	15
2.3.2.2 Process Interaction Via Message Passing	16
2.3.2.2.1 Communication Primitives	17
2.3.2.2.2 Naming Senders and Receivers	19
2.3.2.2.3 Blocking Characteristics of Message Passing Primitives	20
2.4 Parallel Languages	21
2.5 Parallel Computing in a Workstation Environment	23
2.5.1 Processor Mapping	23
2.5.2 Communication Cost	24
2.5.3 Idle Workstations	24
2.5.4 Ease of Programming	24
2.6 Techniques of Parallel Computing Using Processor LANs ...	25
2.6.1 Operating Systems	25
2.6.1.1 Primitives for Interprocess Communication in Unix	26
2.6.1.2 Remote Procedure Calls	27
2.6.2 Support for Workstation Environment in Parallel Languages	28
2.6.3 Utilizing Idle Processors	29
2.6.3.1 Engineering Computer Network (ECN)	29
2.6.3.2 The Butler	30
2.6.3.3 The Worm Programs	30
2.6.3.4 Process Server	31
2.6.4 Special Model for Workstation Environments	32
2.7 Conclusions	33

3.0 Chapter 3: FrameWorks: The Model	35
3.1 Introduction	35
3.2 The FrameWorks Model	36
3.2.1 Modules	37
3.2.2 Templates	41
3.2.3 Input Templates	42
3.2.3.1 Initial	42
3.2.3.2 In_pipeline	42
3.2.3.3 Assimilator	43
3.2.4 Output Templates	43
3.2.4.1 Out_pipeline	43
3.2.4.2 Manager	43
3.2.4.3 Terminal	44
3.2.5 Body Templates	44
3.2.5.1 Executive	44
3.2.5.2 Contractor	47
3.2.6 Sources of Parallelism	47
3.2.7 Processor Binding in FrameWorks	48
3.3 An Example	49
3.4 Choice of Nomenclature in FrameWorks	52
3.5 Critical Review of FrameWorks	54
3.6 Conclusions	56
4.0 Chapter 4: FrameWorks: Formal Specification and Implementation Models	57
4.1 Introduction	57
4.2 Structure of FrameWorks Application	58
4.2.1 Criterion for Replication	58
4.2.2 Frames	59
4.2.3 Links	59
4.2.4 Criterion for Call	59
4.2.5 Structural Constraints on Application Graph	62
4.3 The General Transition Model	63
4.4 Dual Nature of FrameWorks' Processes	64
4.4.1 FrameWorks' Processes as Servers	64
4.4.1.1 Definition of Input Criterion and Client Selection Function	65
4.4.2 FrameWorks Processes as Clients	67
4.4.2.1 Definition of Server Selection Function	67
4.4.3 Body Templates	69
4.5 Implementation Models and Their Communication Protocols	69
4.5.1 Restricted Model	72
4.5.2 General Model	74
4.5.2.1 Activity and TimeStamp Matrices	79
4.6 Concurrency Issues	80
4.6.1 Fairness	80
4.6.2 Termination	82
4.6.3 Deadlocks	85
4.7 Conclusions	87

5.0 Chapter 5: The FrameWorks System	88
5.1 Introduction	88
5.2 Developing Applications Using FrameWorks System	89
5.3 Components of the FrameWorks System	90
5.3.1 Module_Craft	90
5.3.2 Text Editor	96
5.3.3 Configurer and Allocator	96
5.3.4 FW_PP	97
5.3.5 Compilation and Linking	98
5.3.6 Execution Time Monitor	98
5.3.7 Butler	99
5.3.8 View_Graph	99
5.4 Implementation Tools and Techniques	102
5.5 Preliminary Experimental Results	103
5.5.1 Application Independent Results	103
5.5.2 The Distributed Radiosity System	111
5.6 Conclusions	113
6.0 Chapter 6: Enhancements to FrameWorks Approach	115
6.1 Introduction	115
6.2 Variations of the FrameWorks Model	116
6.2.1 Hierarchical Node Organization	116
6.2.2 Automatic Sequential/Parallel Conversion	118
6.2.3 Contractors and Subcontractors	121
6.2.4 FrameWorks in Tightly Coupled MIMD Environments ..	125
6.3 Potential Enhancements to the FrameWorks System	126
6.3.1 Process Migration	126
6.3.2 An Integrated User Interface	127
6.3.3 Better Communication Protocols for LAN-Based Environment	128
6.3.4 Extension and Refinement of Templates	129
6.3.5 Developing Support Subsystems	129
6.4 Tools Used for Developing FrameWorks	131
6.5 Experience With FrameWorks	132
6.6 Conclusions	133
7.0 Chapter 7: Critical Review and Concluding Remarks	135
7.1 Introduction	135
7.2 Comparison with Relevant Research Works	135
7.2.1 FrameWorks as a System for Shared Processing	135
7.2.2 Templates and Related Macro-expansion Based Approaches	137
7.2.3 FrameWorks as a Visual System	138
7.2.4 Comparison with Other Languages, Tools, and Systems	139
7.3 Thesis Summary	141
7.4 Conclusions	142
References	143

A.0 Appendix A: The Overall Structure of the Distributed Radiosity System	154
B.0 Appendix B: Some Example Applications	155
B.1 The Prime Number Generator	155
B.2 Mandelbrot Sets	156
B.3 The Parallel Make System	158
B.4 The Animation System	159
B.5 Some Observations	161

Chapter 1

Shared Processing in a Workstation Environment

1.1. The Growth of Workstation Environments

Computing environments have gone through a vast change during the last decade. The large centralized time sharing systems are steadily being replaced by networks of workstations in commercial, academic, research, and industrial environments. Advances in hardware as well as software technologies have contributed in shaping this evolution. First, developments in VLSI technology have significantly brought down the cost of hardware processors. Second, advances in data communication technology have made possible high bandwidth local area networks that provide access to facilities on the network. Third, peripheral devices, such as high quality printers and plotters, still remain expensive. These factors combined together result in a computing environment in which a number of autonomous processors with private primary memory (sometimes secondary memory as well) are connected together on a local area network. These autonomous processors are called computer *workstations* and this kind of computing environment is known as a *workstation environment*. Access to expensive peripheral devices is still shared among the workstations users. Such an environment provides its users with dedicated and localized computing power which results in a much improved response time. The overall computing power versus system cost ratio is much lower in the case of workstation environments. In addition to providing shared access to expensive peripherals, the high bandwidth local area network also makes possible sharing of information among workstations (e.g. via distributed file systems) and efficient inter-user communication. From the software engineering point of view, workstation environments were greatly influenced by developments in the area of graphical user interfaces. A user interface can present its user with a view of the machine that is more suitable to his needs and capabilities. A large number of the workstations available today feature sophisticated graphical subsystems with interaction devices such as

mice and touch-sensitive screens.

However, a workstation environment is not without a few drawbacks. The system is more complex than a centralized time sharing system. This has its implications on several different areas such as file system design, security, maintenance, and file backup/recovery. One such aspect of the workstation environment is its lack of facilities for load sharing or shared execution of large computations. This lack of shared execution capability is the major concern in this thesis.

1.2. The Problem

With the speed of a single processor approaching device limits, distributed and parallel computing are generally regarded as promising avenues to high-speed computing. In the last decade, there was a rapid increase in the number of multiprocessor architectures that were developed along with their programming models, languages, and programming environments. As described above, the same period saw the replacement of large mainframe-based environments by workstation environments at a rapid pace. Although it has long been recognized that a network of workstations represents a large amount of computing power, the state of the art in harnessing this power is still in a state of infancy. Consequently, large time-sharing systems or expensive parallel computers continue to be used for large jobs that cannot achieve acceptable performance on a single workstation. However, distributed computing in a workstation environment poses several challenges that do not have an analogue in the tightly coupled multiprocessor environment or in the large time-sharing environment:

- 1) A basic principle of the workstation environment is that the facility of shared processing should not restrict other users in the environment. A computation should use only the currently available processors. Therefore, the number of the processors available to a distributed application, as well as their capabilities and physical addresses, may vary from one execution to another.
- 2) Communication costs over the network of workstations are generally higher than

similar costs in the case of tightly coupled MIMD environments. This restricts the types of parallelism that can effectively be implemented.

- 3) In such an environment, expert users may want to exercise control over the assignment of processors because of factors such as ownership constraints and processors' capabilities, while naive users may not like to deal with the details of the underlying hardware architecture.
- 4) Often, satisfying the above three criteria means that a user become an expert in networking or low-level communication packages to utilize the potential parallelism. Usually, users do not like this option.

1.3. The FrameWorks Approach

This thesis is concerned with a high-level model and system, called *FrameWorks*, for writing applications in a distributed environment while keeping in view the above mentioned constraints on the environment and the users. Our initial work consisted of the design and development of a customized distributed animation system [49] based on the concepts of *FrameWorks*. The model described in this thesis, however, is general enough to be employed for a large class of applications. Briefly, the *FrameWorks* model is based on following basic design principles:

- 1) It separates the specification of the structure of the distributed application (communication, synchronization, scheduling, and processor binding) from the program code for the application.
- 2) It uses an extended and modified version of the well known remote procedure call protocol [9] to keep the overall structure of the distributed application close to that of the equivalent sequential program.
- 3) It emphasizes both partitioning and replication for exploiting potential parallelism.
- 4) Graphs are considered important for the user to visualize and understand his/her program.

A computation in *FrameWorks* is represented by an ordinary directed graph. A node of the graph represents a computational unit which is nothing but a set of sequential procedures enclosed in specially designed graphical icons called *templates*. The code in the computational units, except for blocking and non-blocking calls to other nodes, consists entirely of sequential procedures. All other synchronization/communication characteristics, as well as the processor allocation constraints of a node are described via *templates* and an interactive graphical interface. By attaching an appropriate set of templates to each node in a network, most of the common forms of distributed process graphs (such as pipelines, fan-in or fan-out structures, master/slave relationships, and process replication) can be quickly constructed. Templates also provide a novel method for distributing work in environments with varying number of processors. A specific template, called the *contractor*, dynamically contracts out work to idle processors. Thus processes run in the background, taking advantage of idle machines when available and recognizing when machines become heavily loaded. In this way, the user community can be kept happy, while applications profitably use machines that would otherwise be idle.

1.4. Contribution of the Thesis

The thesis is concerned with the development of a high-level computational model and system suitable for shared processing in a workstation environment. In this section, we outline the major research contributions of the thesis:†

- (1) The thesis describes a high-level computational model for distributed computing which is based on an entirely novel concept, called template attachment. The key feature of the template-based approach is that it separates the specification of the program code of a module from its interface, interconnection, and processor allocation aspects.

† The contributions discussed in items (1), (2), (4), and (5) here have also been described, in part, in our earlier publications [98-100].

- (2) To demonstrate the practical feasibility of the model, the thesis describes the implementation of a system, called the *FrameWorks System*. Two separate versions of the *FrameWorks System* have been implemented at the Department of Computing Science, University of Alberta. The implementations use a pool of various versions of Sun-3 workstations (Sun-3/50, Sun-3/60, etc.) which are connected over a local area network using ethernet interfaces. These systems consist of approximately 20,000 lines of source code in C language. The separation of specifications provided by the template-based approach results in several interesting features:
- a) Multiple forms of application partitioning and replication alternatives allow an application to be restructured rapidly and reliably to match the available number of processors.
 - b) In most cases, the complete application can be compiled to run in a sequential or distributed form with only minor (if any) changes on the user's part.
 - c) Processors can be assigned externally to nodes and a user exercises as much control over the mapping of processes to processors as desired. Using the graphical interface, the user can specify the processor assignments completely, partially, or leave it entirely up to the environment.
- (3) The template-based approach of *FrameWorks* is graph-based and, therefore, inherently suitable for implementation as a visual system. To the best of our knowledge, *FrameWorks* is the first visual system meant for structuring distributed algorithms for a workstation environment. The application graph as used by the system makes the parallel structure of the application explicit and provides significant aid in visualizing, documenting, and tuning the application.
- (4) The idea of providing shared processing in the workstation environment has been studied by several researchers. However, *FrameWorks* is the first system that not only keeps track of the available processors in the environment but also offers a conceptual model for developing distributed applications, which is consistent

with the constraints of the workstation environment.

- (5) A specific template of the *FrameWorks* System, called the contractor template, provides a novel method for distributing work in an environment with changing resources. Contractors dynamically contract out work to employee resources. They adapt to the changing constraints of the environment.
- (6) In the past researchers have often debated the pros and cons of utilizing anthropomorphic terminology for describing computational systems. In *FrameWorks*, we have attempted to absorb the essence of the differing viewpoints on the issue. The nomenclature used in *FrameWorks* is almost entirely anthropomorphic, i.e., it is based on entities and structures commonly found in everyday life, e.g., manager, contractor, etc. Although we do have the distinction of being the first system which is entirely based on anthropomorphic terminology, we understand that any quantitative measurement of the impact of this decision would be a rather difficult and certainly a long term process.

1.5. The Scope of the Thesis

The thesis describes a new approach to structuring distributed algorithms. To keep our research work focussed on issues that are central to this approach, and also because of limitations on resources at our disposal, the research work described here is restricted in several ways:

- (1) The system implements only the major aspects of the *FrameWorks* model that are necessary to demonstrate the feasibility of the approach. So far, our goal has been to design and develop a system that allows initial development of distributed applications. Some effort has also been put towards developing a component that would allow performance visualization and tuning. However, our work does not preclude development of other components that would aid activities such as application design, source code management, and debugging.

- (2) Some important aspects of distributed systems, such as fault tolerance, load balancing, optimum processor allocation and scheduling are not part of this work. Although, in a broad way these aspects are related to *FrameWorks*, except for minor differences, the essential problem in each case is hardly different from the general version of the respective problem. Therefore, it is felt that these aspects can best be dealt with in their own respective research areas.

1.6. Organization of the Thesis

Chapter 2 of this thesis reviews some basic concepts of distributed systems. Previous approaches to providing shared processing in a workstation or in a general computer network environment are discussed. Chapter 3 provides an informal introduction to the concepts of our computational model and illustrates its operation through a detailed example. The treatment in Chapter 4 is more formal whereby the structure of a *FrameWorks* application is defined using set theoretic notations. A specification technique called the *General Transition Model* [12,13] is used for specifying the behavior of various templates of the model. Also, two implementation models for *FrameWorks* are proposed and their communication protocols are specified. Chapter 5 describes the implementation of two versions of the *FrameWorks* system that are based on the two implementation models described in Chapter 4. Some preliminary performance data for the two implementations is presented. There are several directions in which further development of the model or the system can take place. Several possible extensions to the model as well as the system are outlined in Chapter 6. In the concluding Chapter 7, a critical analysis of *FrameWorks* is presented and the research work in this thesis is compared with other research works from several different points of view. Finally, we present some concluding remarks about our work.

1.7. Conclusions

Given the continued growth of workstation environments, the general availability of idle computing power in such environments, and the lack of tools for easily

exploiting this computing power, the issue of structuring medium and large grain distributed applications in these environment is likely to continue to attract serious attention of researchers in the coming several years. This thesis describes an early attempt to define a new model for using a pool of workstations for parallel programming purposes.

Chapter 2

Review of Basic Issues

2.1. Introduction

Parallel and distributed computing are, nowadays, regarded as the most promising approaches to high-speed computing. In the past, a variety of multiprocessor architectures have been developed along with the basic tools and techniques that make these architectures usable, e.g., programming models, language compilers, and debugging tools. On the distributed side of the spectrum, several kinds of distributed computing systems have been built ranging from tightly-coupled multicomputer networks to networks of computers connected on local or wide-area networks. However, the work in the two areas differed significantly in terms of their design goals, nature of target application areas, programming models and techniques. The research in parallel computing was primarily aimed to achieve highest possible performance levels. The goals in the area of distributed systems were usually a mix of providing resource sharing, attaining higher reliability through distribution, implementing an inherently distributed system, high-speed computing, and achieving superior structural representation or implementation via distributed structuring. Research in this area was often heavily influenced by the underlying application domains such as operating systems and distributed database management systems.

The advent of workstation-based computing environments has made computer networks quite commonplace. Although the primary goal of these networks was to provide high quality personalized computing along with resource sharing, the notion of sharing originally did not include the computing power of the workstations itself. However, researchers lately have been investigating the idea of using such networks of computers for the purpose of shared or parallel computing. As mentioned in Chapter 1, the present thesis is a continuation of this investigation. In this Chapter, we intend to review the major techniques, tools and systems that are relevant to our research.

The following section defines the scope of the present chapter. The major issues concerning basic interaction techniques in a parallel processing environment are described in Section 2.3. Section 2.4 presents a quick tour of parallel programming languages including a discussion of CSP, DP, and ADA. Section 2.5 describes the state of the research in systems, languages and tools that are specifically directed towards parallel processing on a computer network, especially workstation networks. Section 2.6 summarizes the findings of this chapter.

2.2. Scope Definition

As mentioned above, the subject of parallel computing using a network of workstations draws from rich and varied sources such as parallel processing, distributed systems, and operating systems [1]. Therefore, it is important that the goals and scope of our review be clearly defined and the focus be narrowed to topics directly relevant to the issues involved in this research.

Parallel processing can be looked upon as a part of the more general execution environment called concurrent processing. Although, a precise or standard definition of concurrent execution is difficult to find, it generally means that there is more than one thread of program execution in progress. Concurrent execution implies different things under different circumstances. For example, a number of processes can share a single processor in a time sharing system. This is called multiprogramming. In this case, the operating system multiplexes the processor among a number of concurrent processes. Another environment for concurrent execution is where each process runs on its own processor, or more than one processors are shared among concurrent processes. This is called multiprocessing or parallel processing. A parallel processing environment is characterized primarily by the underlying processor architecture and the method of expressing parallel threads of execution. These characteristics further determine the detailed techniques for interaction among different threads of execution.

A variety of architectural models exist for the purpose of parallel or distributed

processing [5, 62, 87]. The single instruction multiple data (SIMD) multiprocessor systems apply the same instruction simultaneously to different data. In multiple instruction multiple data (MIMD) multiprocessor architectures, different instructions may be applied in parallel to different data. The MIMD architectures can further be classified as shared memory multiprocessors and multicomputers. In a shared memory multiprocessor architecture, a number of processors have access to a global or common memory. Multicomputers on the other hand do not have any shared memory. Communication among processors, in this case, takes place through a message passing network. Multicomputer systems that do not have shared memory and communicate via sending messages over a network are also referred to as distributed computing systems. Based on the characteristics of their message passing network, the multicomputers can further be classified as tightly coupled multicomputers (e.g. Intel's Hypercube architecture and cosmic cube [3]) and loosely coupled multicomputers (e.g., a network of computers connected on a local or wide-area network). According to this classification, a network of workstations connected over a local area network falls into the category of loosely coupled multicomputer architectures.

There are several different ways in which the parallel threads of execution may be specified. Depending upon the programming paradigm, threads of execution may take different forms. In a procedural programming language paradigm, parallelism is most commonly expressed via processes. However, in a object oriented environment, it may take the form of objects; in a functional programming language, parallelism may be expressed in terms of expressions; whereas in a logic programming environment, it may take the form of logic clauses. In some cases, parallelism can also be looked upon as being expressed in terms of groups of statements [70].

In addition, there are other aspects, such as reliability, verification, testing, debugging, and security, that characterize a parallel execution environment. However, keeping in view the scope of our work, and also to contain the size of the document within a reasonable limit, we have refrained from going into these issues. In this chapter, we

are primarily concerned with investigating the state of the art of models for building high performance applications using a network of workstations that are connected over a local area network.

2.3. Review of Basic Techniques for Parallelism

As mentioned above, in procedural languages, parallelism is generally specified in the form of *processes*. Although, there is no universally agreed upon definition of a process, it is generally referred to as a 'program in execution' [33]. Therefore, parallel execution in this case implies parallel execution of more than one process. Logically, each process maintains its own sequential code, data, and state. There are several aspects that may be used for evaluating any model of parallel execution. However, the following two are considered fundamental:

- (1) the specification of parallel execution, and
- (2) the mode of process interaction.

The problem of process interaction can further be broken into subproblems of synchronization and communication. There are a number of schemes proposed for the specification, synchronization and communication of parallel processes. In this section, we review these schemes.

2.3.1. Specification of Parallel Execution

There are three methods of specification of parallel execution that have received widespread attention. Their variants have been used in a number of languages. These are *coroutines* [28], *fork* and *join* statements [29, 34] and *parbegin/parend* statements [36]. The *coroutines* mechanism allows specification of multiple threads of execution in the form of a main program and coroutines. Control of execution may be transferred among coroutines. However, at one time only one of these threads is active. In this way, coroutines do not represent real parallel processing and can be looked upon as "quasi-parallel" processes. Therefore, we shall not consider coroutines any further in this chapter.

Unlike coroutines where only one routine executes at a given time, several routines can be executed in parallel using the *fork* and *join* construct of Conway [29], and Dennis and Van Horn [34]. A fork command is used to initiate a new routine. After the execution of a fork statement, the original routine and the invoked routine execute in parallel. To achieve synchronization, the invoking routine may execute a join statement. At this point the invoking routine may have to delay execution if the invoked routine has not terminated. On the termination of the invoked routine, the original routine continues the execution of the program following the join statement. A well-known system that uses a variant of the fork/join concept is the UNIX operating system [90].

The *parbegin/parend* construct was initially proposed by Dijkstra [36] and is used by the Communicating Sequential Processes language of Hoare [57]. Using the *parbegin* statement a parallel execution can be specified as

parbegin A::statements || B::statements || C::statements parend

Each of processes A, B, and C may be any number of statements including *parbegin* statements or local declarations. This command imposes a one-in, one-out structure on process initiation and termination which may be nested. Execution causes all A, B, and C to execute in parallel and requires all three to terminate before the next command can be executed.

2.3.1.1. Comparative Analysis

There are a number of criteria which can be used to evaluate a notation for the specification of parallel execution. Some of the important considerations for this purpose are dynamic creation/termination of processes, the control structure imposed on process creation and termination, and the creation of multiple instances of a declared process.

Dynamic creation of processes is not possible using *parbegin/parend*, whereas the *fork/join* statement supports the dynamic creation of processes. *Parbegin/parend*

imposes the discipline of a single-entry, single-exit control structure. This makes programs much easier to understand. This is in contrast to the unrestricted ways in which fork and join commands may be used. Although fork and join commands provide a more powerful construct than parbegin/parend, their undisciplined use may give rise to complicated programs where understanding which routines execute in parallel may be quite difficult.

Parallel languages often combine features from more than one mechanism to construct their process creation/termination structure. For example, in Ada [8] the static creation/termination of processes has the single-entry, single-exit structure of parbegin/parend. But like the fork/join statement, it also supports the dynamic creation of processes by which new threads of control can be created at run-time. However, all tasks must terminate at the end of the unit in which they are initiated. Therefore, the process execution of Ada can be described as a multiple-entry, single-exit structure.

Yet another criteria for judging these constructs is the capability of activating more than one instance of a process. Concurrent Pascal [55], for example, allows creation of multiple instances of a process whereas others, like Distributed Processes [53], do not.

2.3.2. Interaction among Parallel Processes

In any parallel processing environment, the interaction among processes can be described as consisting of contention and/or cooperation. Parallel processes compete with each other for mutually exclusive access to resources whose supply may be limited, or whose unrestrained use may result in some conflict. Also, processes are required to follow certain protocols in order to exchange information or to compete with each other for shared resources. Two basic mechanisms that support such interactions among processes are known as communication and synchronization. Communication protocols allow processes to exchange information and thus influence each other's course of execution. Synchronization is necessary for effective utilization of

shared resources, and ordering of events, as well as communication. For example, a process attempting to write in a buffer should be delayed until there is space in the buffer. Such an event should be indicated to the process by another process that reads the buffer. Similarly, a process that attempts to read an empty buffer should be delayed until the arrival of data is indicated by a process that writes into the buffer.

The mechanisms for communication and synchronization among processes are influenced greatly by the underlying architecture on which the system is implemented. In a centralized system, this is generally realized by sharing information in a globally accessible part of the memory. In a distributed system, they are achieved by means of message passing. In the following sections we review communication and synchronization mechanisms. The emphasis of the review is on message passing environments, because of their suitability to distributed systems.

2.3.2.1. Process Interaction in a Shared Memory Environment

In a shared memory environment, processes have access to some global part of the memory. Interprocess communication can be achieved by reading or writing variables in the global memory. For example, one process may set the value of a variable and the other process may read it. Shared variables can also be used for process synchronization. *Busy Waiting* is one of the early mechanisms proposed for process synchronization in a shared memory environment. However, designing correct process synchronization protocol becomes tricky and complicated when more than two processes are involved. Also, busy waiting wastes processor time. *Semaphores* of Dijkstra [36] can be used to design process synchronization without wasting processor time. However, because of the low-level nature of semaphores, difficulties arise when some processes are not well behaved. This may happen due to an honest programming error or an act of deliberate mischief. Realizing these difficulties with low-level synchronization primitives, Brinch Hansen and Hoare designed high level process synchronization schemes, such as *conditional critical regions* (CCR) [54, 56, 59], and

monitors [54,58]. *Path expressions* of Campbell and Habermann [19] provide an alternative high-level approach for specifying in a non-imperative manner the concurrency constraints on the operation of a resource. Although, shared variables are more suited to centralized and shared memory systems and are generally not used for communication in distributed systems, there are some distributed systems that do use variants of shared variables. For example, Linda's tuple space implements global data structures that are accessible by any process in an application [20].

2.3.2.2. Process Interaction Via Message Passing

A contrasting approach to shared memory is provided by message passing systems which are almost always used for distributed systems, although several systems use them for communication among local processes as well. The two basic primitives for message passing are:

send (destination, message);
receive (source, message);

A send operation moves the data contained in a message from its sending process to the destination process. A receive operation is used to fetch data from the process specified in the source. Generally, the source and destination are specified as processes, however, some systems allow specific statements within a process to become a source or destination as well. Message passing is an effective method for communication as well as synchronization. Communication obviously takes place since data is moved from one process to another. Synchronization can be implemented because a message can be received only after it is sent. This puts constraints on the order in which events take place. As in the case of a shared memory environments, message passing schemes have to deal with the two basic issues of:

- (1) what communication primitives are used, and
- (2) how synchronization is achieved.

In addition, message passing systems have to deal with the issue of designating

senders and receivers. In the following sections we survey some important methods for dealing with each of these problems.

2.3.2.2.1. Communication Primitives

The most common schemes for sending and receiving messages are those where each process that wants to send or receive a message explicitly names the recipient or the sender process.

```
send (process, message);
receive (process, message);
```

A broadcast may be implemented as several parallel sends executed by the sending process. The send and receive primitives mentioned above are symmetric in the sense that each communicating process names the other. Several systems provide a receive primitive that allows a process to receive a message from any process [24]. This may be specified as

```
process = receive (message);
```

where the variable **process** gets set to the identity of the sending process. Several systems also provide features by which a process can query whether there is any message from any other process or a process could have a peek at the message without actually reading it [81].

The *client-server model* represents another common situation where a higher level protocol than the basic send/receive can be quite useful. In this model, both client and server processes follow a well-known protocol. Depending on the situation, the protocol may be symmetric or asymmetric. In an asymmetric protocol one side is always recognized as the master and the other side as the slave. In a symmetric protocol either side can play the role of master or slave. In either case, while accessing the services, there is always a "client" process and a "server" process. The server process, normally listens to a known address for service requests. The client side requests the service. After a connection is established, data transfer may begin between the two

processes according to the implementation of the application.

Depending upon the particulars of an implementation, a server may queue up the calls while it is servicing a call. This may happen, for example, for a printer server when there is only one printer for serving the printing requests. Another possibility is that the original server may create a new process which then serves the client while the original server continues to listen for more service requests. Some systems have a special service server which listens for any kind of service requests. This has the advantage that too many idle servers do not clog the system. The service server identifies the service requested by a client, creates the appropriate server, and "splices" together the client and server processes thus completing its role in the transaction. Though, this scheme has the overhead of having an extra server, it is quite attractive from several points of view. First, the service server provides a single point where all the available services can be requested. Second, it can be used for other purposes, such as authentication. It also reduces the cost of a client-server interaction.

Although the particulars of interaction between the client and the server side depends on the implementation, *rendezvous* and *remote procedure calls* represent two distinct constructs based on the client-server model [1, 5, 86].

In a rendezvous, the client process calls an entry defined in the server process. When the server is ready to process the request, it issues an accept statement. The caller then passes input data to the server and waits for the output data. The input data is processed by the server and the results, if any, are passed back to the client. The interaction between client and server is fully synchronous and the side that is ready to interact has to wait for the other side.

In a large number of cases the interaction consists of client executing a send followed by a receive, and the server executing a receive followed by a send. Therefore, higher level protocols that directly support this type of interaction have been proposed [9, 53, 74, 115]. Interaction using a remote procedure call (hereafter called RPC) takes place as follows: a client process calls a procedure which is most likely executed on a

remote machine. The calling procedure then transfers the parameters to the remote procedure. The execution environment in the calling procedure is suspended. The remote procedure receives the input parameters, executes the procedure and produces the results that are passed back to the calling procedure. The paradigm of remote procedure calls has many attractive features. First, it provides a high-level alternative to using low-level message passing primitives. Second, it has clean and clear semantics. This makes the RPC mechanism easy to understand and use. Third, the concept of a RPC is based on the general concept of procedure calls which are used locally within a process as well.

2.3.2.2.2. Naming Senders and Receivers

The most straightforward method of naming sender or receiver is the direct naming scheme. This scheme specifies a one-to-one channel of communication. However, such a specification is not well suited for several applications such as client-server models. In such applications one or more servers offer some service. When a client process requests the service, an available server process does the job and, if needed, replies to the client. Mailboxes have been proposed to provide the naming of communication channels for this purpose. A mailbox can be viewed as an object into which messages can be placed by the sending processes and retrieved by the receiving process. Although mailboxes provide communication channels that are many to many, their implementation where a message may be processed by a single receiving process, is expensive [45]. A message sent to a mailbox is relayed to all potential receivers. When a site receives it, all other sites must be informed of the event. PORTS by Balzer [7] implement a special case of mailboxes. In this case several senders can send messages to a port but a single process receives them. Ports provide an inexpensive solution to the many clients-single server situation.

An important aspect of naming is whether communication channels are statically defined at compile-time or are dynamically assigned at run-time. An example of

dynamic channel specification is the implementation of pipes and filters in the Unix operating system [90]. Dynamic channel specification allows programs to communicate along channels not known at compile-time.

2.3.2.2.3. Blocking Characteristics of Message Passing Primitives

The blocking characteristics of message passing primitives largely determine the nature of the synchronization mechanisms that can be used. Also, they influence the amount of parallelism that can be exploited in a concurrent system. Blocking characteristics specify whether processes intrinsically block as a part of the operation of sending or receiving a message. First, a process executing a message need not wait at all. This is called asynchronous send or *no wait* send. This means that a sending process can run arbitrarily far ahead of the receiving process, requiring a buffer of virtually unbounded capacity. The situation can be cured by putting a bound on the number of messages (or total size of the messages) that can be buffered. This allows the sending process to proceed ahead of the receiving process but not to an arbitrary extent. After a limit is reached, the sending process may block, or may receive a return code indicating that the buffer of the receiving process is full.

More commonly, the sending process waits until the message has been received by the target process. This is called synchronous send [57]. A different but useful form of send is used while executing remote procedures. In this case the sending process may block until the procedure has been executed by the remote process and a reply has been received [53]. Evidently, it is possible to think of a send process that waits till the parameters have been received by the remote procedure but does not wait for a reply [2].

Similarly, both blocking and nonblocking forms of receive are useful. If data from a specific process is required, the programmer codes a blocking form of receive. The process then waits for data to arrive. Unfortunately, the blocking form of receive can lead to a deadlock when two processes are waiting for data from each other. The

nonblocking form of receive is used for an asynchronous input. A process expecting input may continually execute this form of a receive to check if there is any message. If there is a message, it may process it, but if there is no message, some other action may be taken. Some languages allow receive statements which receive only messages that satisfy some condition [40]. This is a form of looking at the messages without actually receiving them.

Blocking send and receive provide a straightforward means of implementing synchronization. Achieving synchronization using nonblocking send and receive requires busy waiting which is wasteful. Dijkstra's guarded commands [37] provide a way to achieve the nonblocking effect while using synchronous message passing.

2.4. Parallel Languages

Having discussed the basic concepts of parallel programming, we turn our attention in this section to parallel programming languages. PL/I and ALGOL 68 are examples of two early languages that addressed concurrency issues. During the last decade, several languages for parallel programming have been proposed. Not all of these languages enjoy widespread use. Some of them have not even been implemented, while others were designed with the aim of getting some insight and experience with a particular language construct or a combination of them. Mechanisms of parallel programming languages can be classified according to how basic building blocks, such as described in the previous sections, are combined. First of all, we can classify parallel languages according to whether they are based on the notion of shared variables or message passing.

Concurrent Pascal was proposed by Brinch Hansen [55] for concurrent programming in a centralized environment. It has three basic building blocks for constructing programs: process, monitor, and class. A process is defined by its private data, sequential program and access rights to other shared data. Processes can communicate with each other only via monitors. A monitor defines a shared data and all the operations

that can be performed on it. These operations are called monitor procedures. A class is a system component that cannot be called simultaneously by several components, but has data structures and procedures like monitors. The benefit of having a class as a separate building block is that a class does not have to schedule simultaneous calls at run-time. Modula [116] is another language that is based on the concept of monitors. However, in Modula programmers can bypass monitors and access shared variables directly. In that respect, Modula is less restrictive than Concurrent Pascal.

The Linda language by Gelernter and Carriero [20] uses the concept of a *tuple space* for interprocess interaction. Conceptually, the tuple space is a globally shared memory, however, it may or may not be implemented using physically shared memory. Elements of the tuple space are similar to Pascal records or C structures. Processes use atomic operations that allow them to simply read a tuple (*read* operation) from the tuple space, read and delete a tuple (the *in* operation) from the tuple space, or add a tuple (the *out* operation) to the tuple space.

Communicating Sequential Processes (CSP) by Hoare [57] and Distributed Processes (DP) by Brinch Hansen [53] are two significant examples of languages based on message passing. Processes, guarded commands, and primitive input-output statements form the basic building blocks of CSP. DP is based on procedure calls. In DP, processes communicate and synchronize via procedure calls and several different versions of guarded commands. Finally, interprocess communication in Ada is a mixture of DP's procedure call and CSP's input-output statements. Concurrency in Ada is expressed in terms of processes (Tasks). Processes communicate using a mechanism, the rendezvous, involving a caller process and a server process. Communication is synchronous and blocking in nature. A caller process calls an entry defined in a server process. When the server is ready to communicate, it issues an accept statement. The caller then passes input data to the server. The data is processed by the body of the accept statement and results, if any, are passed to the caller. To control non-determinism, Ada provides a select statement which is based on Dijkstra's guarded

command. The select statement allows servers to accept entry calls in a flexible manner.

2.5. Parallel Computing in a Workstation Environment

A workstation environment has some special characteristics that make parallel computing using a network of workstations much different than using any other tightly coupled multiprocessor. Few conventional programming languages address these differences. The following subsections examine these differences in detail.

2.5.1. Processor Mapping

Providing shared processing in a workstation environment should not infringe on the basic fact that generally processors in such environments are owned by individuals and these individuals should not feel restricted by the presence of a shared processing facility. Therefore, the number of processors and their physical addresses and capabilities may vary, not only from one execution to another, but also within a single execution. Most of the parallel programming languages hide the architectural characteristics from the programmer, who views his distributed program as being run on a virtual machine whose structure is given by the process structure of the program. This approach has certain advantages. First, some architectures are significantly complicated and, therefore, it seems proper that a user of a high-level language be kept isolated from its details. Second, it makes the language more generally applicable. Therefore, in such languages the run-time environment handles the assignment of processes to processors and the run-time scheduling of processes. However, there are performance realities associated with this, since there is an inherent difference in efficiency between invoking an operation that is local and one that is remote. The issue of assignment and scheduling of processes is a difficult one and depends largely on an application's characteristics such as, cpu time requirements, interprocess communication costs, and existence of precedence relations among processes. It cannot, in general, be handled satisfactorily without the programmer playing some role in this

decision process [5]. As Jones and Schwartz note in their study of the use of *CM** [66], architectural realities in a distributed system cannot be masked from the user without severe performance penalties. A programmer should be able to specify performance bottlenecks of his application, and his preferences regarding the mapping of processes to processors. Equally important in this case is the ability to modify the mapping without any modification to the module itself. The implementors of Eden [11] found it valuable to be able to specify the location where an object executes although objects are prepared with the philosophy of keeping them independent of their location.

2.5.2. Communication Cost

Interprocess communication costs in a loosely coupled workstation environment are much higher than for tightly coupled multicomputers (e.g. Intel's Hypercube). Higher communication costs make exploitation of fine-grained parallelism unattractive. The low-level primitives that allow the exploitation of fine and medium grained parallelism, can also be used to construct applications that use coarse grained parallelism. However, a tool or system designed specifically for loosely coupled environments can be simplified using the fact that it is going to be used primarily for exploiting coarse grain parallelism.

2.5.3. Idle Workstations

Most of the computing power that can be tapped by a shared processing facility in a workstation environment lies in the idle workstations on the network. Therefore, a desirable characteristic of any tool or system for parallel processing in this environment would be to identify and use the idle processors on the network.

2.5.4. Ease of Programming

Finally, distributed programming using a network of processors is no longer the sole province of experts in operating systems or other communication areas. It has

become important to programmers of all kinds of applications, such as large scale scientific and commercial computations, graphics, image processing, robotics, etc. Therefore, in addition to the powerful low-level tools, the environment should also support high-level methodologies that are easy to understand and easy to use by programmers who are not experts in distributed computing.

2.6. Techniques of Parallel Computing Using Processor LANs

In general, there is a paucity of high-level models that address the above requirements of parallel computing in a workstation environment. In this environment, programmers generally rely on low-level tools provided by the operating system. Recently, however, some parallel languages and tools have been proposed that attempt to address one or more of the above-mentioned requirements of the workstation environment. In this section, we review the high- as well as low-level tools, languages and systems that have been proposed or used for shared processing using local-area network of computers, especially workstation LANs. Generally, these tools fall into one of the following four categories:

- 1) message passing primitives provided by the operating system,
- 2) parallel languages with support for the workstation environments,
- 3) specialized tools aimed at utilizing power of idle workstations, and
- 4) parallel programming models specially tailored for workstation environment.

In this section we shall discuss each of these approaches in detail.

2.6.1. Operating Systems

There are two different approaches to managing communication and sharing in a network environment [109]. They are called Network Operating Systems (NOS) and Distributed Operating Systems (DOS). In NOS, each host of a computer network has a local operating system that is independent of the network. The sum total of all the software added to each local host in order to communicate or share resources is called

a NOS. The NOS's are characterized as being built on the top of existing operating systems of the individual hosts. The environment provided by Sun workstations working under Unix is an example of a network operating system. This approach has the advantage that existing software can be utilized.

In the case of the Distributed Operating Systems (DOS) approach, the individual operating systems are discarded, and a single homogeneous operating system is built for the entire network. This approach requires much more effort than developing a NOS but provides more conceptual homogeneity. A DOS is developed keeping in mind the requirements of the network from its inception and it is designed to manage resources in a global fashion. Over the last few years several distributed operating systems have been developed [23, 85, 105, 113]. Both types of operating systems generally provide features such as distributed file systems, creation and destruction of processes on remote hosts, and interprocess communication. Such facilities can be used for developing large distributed applications. In this section, we describe two types of message passing primitives that are provided by Sun's NOS. Most other operating systems support similar primitives.

2.6.1.1. Primitives for Interprocess Communication in Unix

Sockets in Unix provide a good example of the low-level communication primitives that can be used by an application programmer to develop a distributed application [71]. A socket represents one endpoint of a communication channel to which an address is bound. The format of the address depends upon the communication domain. There are two domains currently supported: the Unix domain, and the Internet domain. A specific socket can be used only for a given domain. Also, there are several kinds of sockets, such as stream sockets, sequenced packet sockets, datagram sockets, and raw sockets. A particular implementation may not offer all kinds of sockets. The type of a socket implies the nature of the service offered by it. For example, a stream socket provides reliable, duplex, sequenced data stream whereas the data transferred by

datagram sockets does not guarantee whether messages will be delivered in the order in which they are sent, or whether they will not be lost, etc.

There are a number of system calls that can be used to build the required communication structure. The *socket* call creates a socket and returns a socket descriptor. A name can be bound to a socket descriptor by a *bind* call. Other sockets use this name to address the given socket. With a bound socket it is possible to rendezvous with an unrelated process. This process is usually asymmetric with one process being a "client" and the other being a "server". The client requests services from the server by initiating a "connection" to the server's socket using a *connect* call. The server, when willing to offer its advertised services, issues a *listen* call at some specified address. Finally, the server uses the *accept* system call to accept an individual connection. When a connection is established data may begin to flow using calls such as *read* and *write*. To terminate a connection, a *close* system call may be used.

2.6.1.2. Remote Procedure Calls

A typical example of the implementation of a RPC protocol is provided by Sun. It provides three levels of interfaces for using the RPC mechanism. At the highest level, all the communication details are hidden from the user; the user simply makes a call to one of the system-wide available remote procedures. The procedure calls *rwho* or *runtime* are examples of this kind of use of the RPC mechanism. At the second level, a user can use the primitive *registerrpc* to register a remote procedure for system-wide use, and *callrpc* for calling a remote procedure. At the lowest level, one can explicitly manipulate sockets that transmit messages. This is meant for applications that need to alter the defaults of the routines.

Although the RPC protocol provides a much needed alternative to the use of low-level message passing mechanisms, such as sockets, not all kinds of applications can be directly modeled using this mechanism. For example, several mechanisms require more than one exchange of messages between the client and server processes.

Clearly, a more flexible model of message transfers is needed for such applications.

2.6.2. Support for Workstation Environment in Parallel Languages

An important criterion for designing a language for developing applications in a workstation environment is whether the mapping of user processes to processors will be under user control. Although this responsibility can be handed over to language compiler, run-time support subsystem, or the underlying operating system, it rules out application specific or customized environment specific mapping strategies. As pointed out by researchers in the past, this may have severe performance penalties [2,11,66] because there is an inherent difference in efficiency in invoking a local operation and a remote operation. In this section, we review two languages that allow the programmer to select the processor where a certain process would run.

The StarMod language [30] distinguishes between processes on the same processor and processes on different processors. In StarMod, a module contains one or more processes and, optionally, variables shared by those processes. Synchronization within a module is provided by semaphores. Processes within a module are located on the same processor and may interact via shared variables. Processes in different modules communicate via message passing.

The Synchronizing Resources (SR) system [2] allows the programmer control over mapping the processes to processors. SR uses the notion of *resources*, which may have several processes. The location of a resource may be specified at the time of its creation. Processes within a resource may synchronize by the use of semaphores, whereas processes within different resources interact via different forms of message passing operations. SR has been implemented to run on a collection of Suns and Vaxes, and the Encore Multimax multiprocessor.

An object based language, Emerald [10], allows the programmer to optionally control the location of an object. For example, objects that communicate often may be put on the same processor, or replicated objects may be assigned different processors.

Objects may also be moved from one processor to another during its lifetime.

2.6.3. Utilizing Idle Processors

Generally in a workstation environment, even during the busy times of the day, there are a significant number of idle workstations [111]. Although, one can utilize remote processors via remote procedure calls, the mechanism discussed above is not capable of selecting a workstation based on some criterion. An additional level of control is needed for this purpose. There have been only a few attempts to address this requirement. The majority of such attempts give the user the facility of executing his program on a remote processor, which in some cases is selected by the user and in others is selected by the system based on its selection criterion [23, 52, 63, 80, 94]. These systems vary widely in their abilities regarding dynamic movement of a remote process, management of required files, and handling of applications that consist of multiple remote processes. These approaches may also be classified according to their underlying software environments. Several of these systems are based on environments where each workstation has its autonomous operating systems and local file systems whereas others have a distributed file system. Yet another set of approaches are based on distributed operating systems. The following subsections provide a brief review of these approaches.

2.6.3.1. Engineering Computer Network (ECN)

The Engineering Computer Network (ECN) at Purdue University [63] was one of early systems that provided protocols for the remote execution of commands. The system offered several application level protocols to provide capabilities of virtual terminal access, file transfer, and a remote execution environment. The system provides protocols that support the execution of a command on a specified remote host. However, more noteworthy are protocols that exploit idle computers on the network. Since the network has several different types of computers, the computers and user jobs are grouped into classes in order to decide where a specific job can be processed. Before

starting remote execution of a command, the system also transfers necessary files to the remote host.

2.6.3.2. The Butler

The Butler system [80] is implemented as a set of programs running on the Andrew workstations. It is implemented without any changes to the kernel. Also, applications that use Butler do not require any change. When a machine is donated to the pool of available machines, it runs a program called *butler*. This program is responsible for adding and removing the machine from the global free machine registry, and for handling requests for remote execution on that machine. Butler allows only one user on a machine but a user can have several processes on the same machine. A user can run a job using this facility by using the command

rem command

The *rem* program on a client's machine first checks the registry to find a free machine. The registry returns the name and network address of the potential candidate machine. The *rem* program has to confirm the availability of the candidate machine by contacting it. If the machine is available, its *butler* removes it from the registry. The *rem* program then sets up the execution environment, invokes the program and waits for it to exit. When the client is done, it releases the machine. The local *butler* process then returns it to the free pool of machines. If the machine is reclaimed by its owner during the remote execution of a program (by logging in at console), the *butler* first warns and then kills any guest processes that may be running, and removes the machine's name from the global registry. A serious limitation of the Butler system is that at one time a *butler* can be used to run only one program on a single machine.

2.6.3.3. The Worm Programs

The worm programs of Shoch and Hupp [94] are experiments in the development of multimachine programs. A worm is a computation that is done on one or more machines. The programs on individual machines are described as segments of a worm.

Each segment carries a number that indicates the total number of segments in the program. The segments in a program communicate among themselves for passing data or keeping track of each other's existence. If a segment fails, a new idle machine must be found. Each segment contains some code for initialization and for communication with other segments. The segments work jointly to perform the computation and at the end of the computation the remote hosts are released.

The main aim of the worm programs was to conduct some experiments in distributed computation using idle workstations to assess the feasibility of such computations. Therefore, the research did not come out with any conceptual model for doing such computations nor did it suggest any particular system or language model. However, it did develop several multimachine applications to gain insight into the related issues. The authors concluded that they had all the basic tools to experiment with the development of multimachine applications that required dynamic resource allocation and movement of processes from machine to machine.

2.6.3.4. Process Server

The Process Server [52] is built under the Cedar computing environment of Xerox for providing remote execution of programs on idle machines. Three types of entities together make up the Process Server: the client, the server, and the controller. Any Cedar workstation can be a Client and can ask for computations to be performed. A group of machines together form a cluster. Each cluster has certain Server machines. The Controller itself is a distinguished server and is responsible for managing the cluster. A request for a process server comes in the form of a package. A package implements a set of commands.

The request for execution begins with the Client which contacts its Controller for a machine. The Controller selects a machine from its pool for the Client to use. This selection is based on the load reported periodically by each Server to its controller. After the selection, the machine identifier is returned to the Client. The Client then

contacts this Server which fetches the required code via the file server and executes it. The request running on the Server is free to FORK and JOIN, and to make further requests inside the Process Server. Eventually the package finishes execution and call to the Server as well as the Client returns.

The Process Server can be used to run distributed applications. However, the programmer codes the distributed application by explicitly ~~using~~ parallel computation using the basic service provided by the Process Server ~~and also~~ coordinates the execution of the parallel processes.

2.6.4. Special Models for Workstation Environments

There are few programming models that are specially designed to exploit computing power of a network of workstations. At this stage, most efforts in this area are only of exploratory nature. In this section, we describe two such research works. Shoja and Gurr [95] study the problem of using a network of IBM PC/XT workstations for parallel computing. The workstations are connected by a 10 Mbps Ethernet. The study discusses the issue of decomposing a consumer-producer based task into parallel executable sub-tasks that synchronize using a mechanism similar to rendezvous. The study also proposes a heuristic model for determining optimum number of workstations based on the grain size of the decomposed modules.

Jagannathan et al. [65] have proposed a programming model for exploiting coarse grain parallelism in applications. In this model, an application is composed of a relatively small number of modules. Each module is an implementation of a function written using a conventional sequential language, such as C with a few syntactic modifications. The interaction among these modules is specified using a directed-graph language (DGL). An application in this model is expressed as a data dependency graph whose nodes denote functions and the edges denote data dependencies between the modules. The incoming edges of the node represent the data input to the module whereas the outgoing edges of a node denote data output from the module. The run-

time system manages the parallel execution of an application. A prototype of the system has been implemented using a network of Sun workstations. Some example applications, such as matrix multiplication and a make program, have been implemented using the system. Results from these experiments indicate that worthwhile performance speed-ups can be obtained.

2.7. Conclusions

The low-level communication primitives are quite powerful for structuring distributed applications. The use of such primitives, however, is by no means trivial and closely resembles performing I/O at the physical level. Since, a large number of users who want to distribute their applications do not possess expertise in the use of low level primitives, these features are not very useful for them. The task of an application programmer is greatly simplified when a remote procedure call protocol is available. However, the approach of using remote procedure calls is deficient in several ways [100]. First, it is quite difficult, if not impossible, to use RPCs for applications that have a large number of interacting processes. With RPCs, it is not possible to directly program applications where the communication consists of several messages from each interacting process. Also, they cannot be directly used to design processes that communicate with more than one processes in order to do their computations. Third, RPCs offer little choice in selecting the processor where the call is going to be executed. Therefore, such protocols are not suitable for environments where the number of processors available for an application and their addresses are likely to change during different executions of an application. A high-level manager is needed for such situations. To summarize, the amount of flexibility and power offered by such protocols is quite limited for the purpose of large applications.

The intriguing structure of worm programs showed the feasibility of developing dynamically expanding and shrinking programs that utilized idle processors. However, no conceptual notation or methodology for this purpose was proposed. The Purdue

ECN system and Process Server have the feature of selecting a server from a pool of machines based on its load average. The Butler system, on the other hand, selects machines from a pool of processors that may be claimed by a user who has priority in using the processor.

The issue of sharing processors for the development of large distributed applications is still by and large unexplored. Although the facilities provided by the Process Server can be used for acquiring more than one processor, the Process Server does not offer any facility that would allow graceful addition or removal of a processor from an application. A system that attempts to allow such addition and removal of processors will have to work in conjunction with a methodology that gives flexibility in terms of the required number of processors. In the following chapter we present a novel approach towards developing distributed applications in a workstation environment.

Chapter 3

FrameWorks: The Model

3.1. Introduction

The discussion in the previous two chapters brings out several interesting characteristics of the workstation environment. First, a network of workstations together represents an amount of computing power which may be comparable to the computing power of one or more supercomputers. Second, these workstations normally belong to individuals, departments, and laboratories who have priority in using these workstations. Third, at any point of time there are a significant number of idle workstations. In other words, there is always a varying but significant amount of computing power on such networks that remains unutilized. These facts make the idea of distributed computing in a workstation environment quite attractive. However, as we have previously pointed out, there are several environment and user related problems that need to be overcome. As described in Chapter 2, researchers in the past have proposed specialized commands, tools, and languages that facilitate shared processing in a workstation environment. However, it is evident that there is still room for improvement in the degree to which these approaches address the requirements of a user, or meet the constraints imposed by the underlying environment. In fact, none of the *FrameWorks'* predecessor systems have been able to provide a high-level model that facilitates structuring of distributed applications while keeping track of idle processors on the network.

In this chapter, the *FrameWorks* model is introduced, which is well suited to developing parallel applications in a distributed environment. The model is based on an extended form of the well known remote procedure call (RPC) protocol [9], which is further modified by a novel concept called, *template attachment*. In *FrameWorks*, the specification of a distributed computation is separated into a high-level declarative part and a low-level procedural part. A distributed computation is expressed as a

directed graph. A node of the graph is constructed using high-level declarative primitives called templates. Each template contributes to the overall synchronization, communication, and scheduling behavior of the node. A selection of templates is available from which a distributed application with the desired behavior can be rapidly built. At the lower level, sequential procedures are attached to the nodes of the computation graph. The approach is intended to provide a model of distributed computing that is easy to use, flexible, and provides the user with the desired amount of control in using idle processing power in the environment. The present chapter provides an overview of the basic concepts behind this model.

The rest of the chapter is organized as follows. The following section outlines the basic concepts of the model. The concepts of the low-level procedural parts, called *modules*, and the high-level declarative parts, called *templates*, are explained. Section 3.3 contains a detailed example from the distributed graphics domain that illustrates how a programmer uses these concepts to develop a distributed application. The names of various design primitives, as well as the names of various components of the *FrameWorks* system, have been based on structures or organizations found in everyday life. This is known as anthropomorphic programming. A brief note on the basic ideas of anthropomorphic programming, and arguments for or against its use are presented in section 3.4. A critique of the *FrameWorks* model, summarizing its scope, strengths, and limitations is given in section 3.5. Finally, section 3.6 contains some concluding observations.

3.2. The FrameWorks Model

We assume an environment where workstations are connected over a local area network and run under a distributed or network operating system that provides basic features for remote process spawning and supports a message passing mechanism among these processes [23, 71, 85, 113]. We also assume the existence of a distributed file system such as the Sun network file system.

The overall organization of a distributed program in *FrameWorks* has striking similarities with the organization of a sequential program. The design of an application program is, in fact, unaffected whether it is intended for sequential or distributed execution. As mentioned earlier, in *FrameWorks* a distributed application is represented as a directed graph, called *application graph* (Figure 3.1). Nodes in this graph represent computational units called *modules*. From a programmer's point of view, each of the modules consists of a set of sequential procedures that interact with other modules via an extended form of remote procedure call [9]. However, the finer details, such as how the input or output interface of a node is constructed, or how static or dynamic replications of a module are managed, are specified using a given set of graphical symbols called *templates*. As shown in Figure 3.2, each node is assigned an input template, an output template, and an optional body template. In this way, the programmer simply specifies the required behavior of a node. The system then automatically generates the necessary low-level code. In many ways, this is analogous to programming with abstract data types, that provide a well-defined means for manipulating data structures while hiding the underlying implementation details from the user.

3.2.1. Modules

Briefly, an application consists of a set of modules. Each *module* consists of a set of procedures exactly one of which is specified as the *entry* procedure. As explained later, the entry procedure of a module can be called by other modules in the application in a manner similar to local *C* procedure calls. In addition to the entry procedure, a module can have *local* procedures that can only be called from within the module. There are no common variables among the modules. Exactly one module in each application contains a special procedure called the *main* procedure. This module is called the *main* module. The *main* module may or may not have an entry procedure. The *main* module initiates the execution of the application.

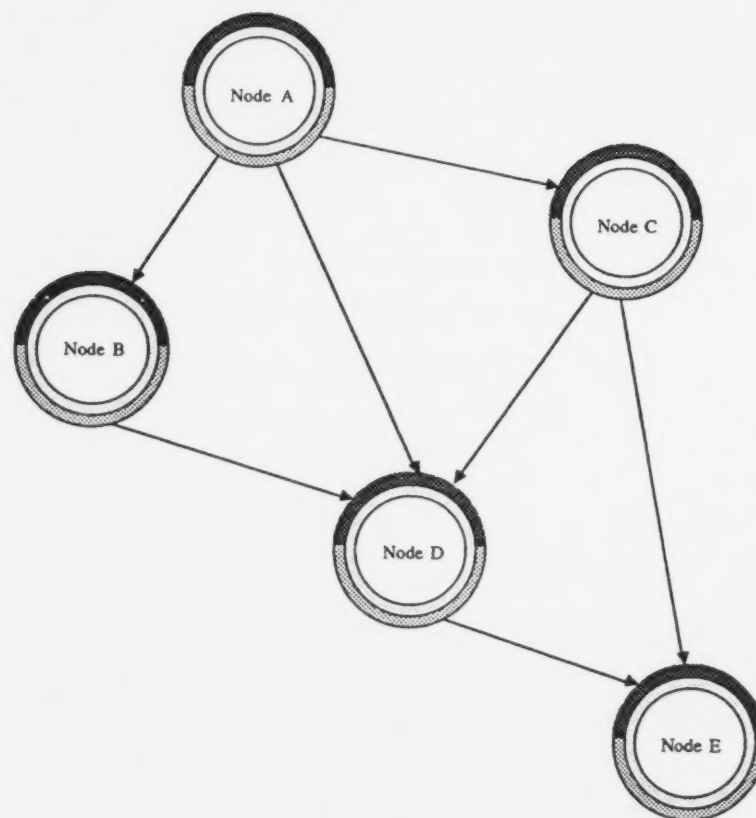


Figure 3.1. Organization of an Application

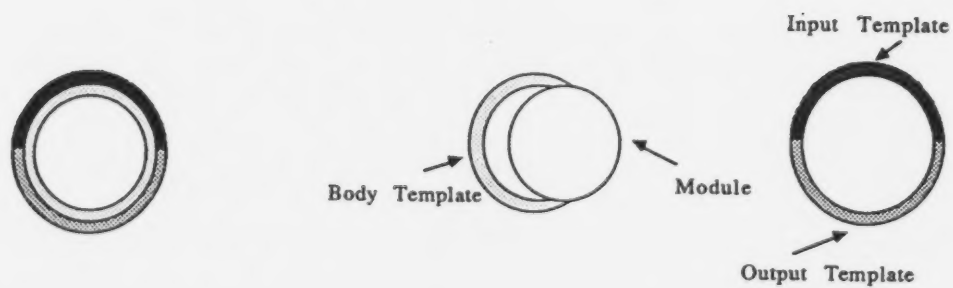


Figure 3.2a. A Node

Figure 3.2b. Parts of a Node

Figure 3.2. Anatomy of a Node

The input and output of modules is in the form of structured messages called *frames*. A frame is similar to a Pascal record or C structure except that pointer type variables are not allowed. For every module, the programmer declares an input frame (a structure containing all the input parameters needed for a *call*) and, if necessary, an output frame (containing all the *reply* or output values returned). The structure of the *main* module as well as other modules is illustrated in Figure 3.3.

FrameWorks augments a programming language (C in our case) with the *call* statement, which has the same semantics as a procedure call, but in fact is translated by the system into communications with another process. A module *X* can call another module *Y* only if there is a directed link from the node enclosing *X* to the node enclosing *Y* on the application graph. The call is made by the simple

call Y(input);

statement, with input being the input frame. With no return value expected, the semantics of such a call are that *Y* is invoked asynchronously, without *X* waiting for its completion (*non-blocking* communication). Synchronous communication is possible using the blocking version of *call*

output = call Y(input);

where output is the return frame from *Y*. The calling routine, *X*, is suspended until the reply (output) from *Y* is received. The called routine replies by executing a

reply(output);

statement.

Each module can serve as a client as well as a server. The execution of an application is initiated by the *main* program of the main module which starts the chain of remote procedure calls among modules. Every other module waits to serve any call to it. When a call is received, it executes its entry procedure to serve the call. During the service of a call, a module may act as a client and call other modules. Unlike sequential programs, the main module may continue to exist after the termination of its main program. At this point the main module behaves just like any other module and

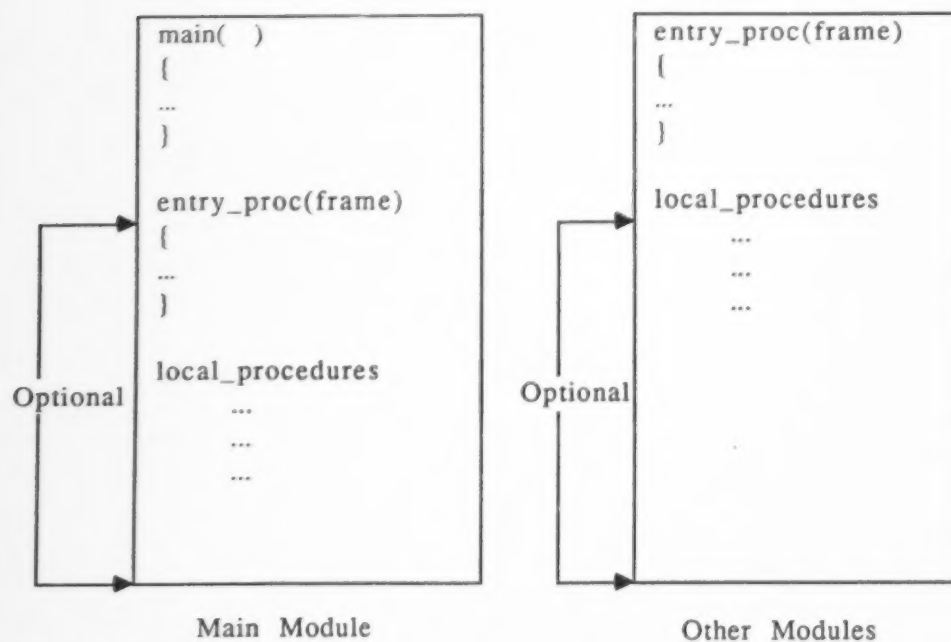


Figure 3.3. Structure of Modules

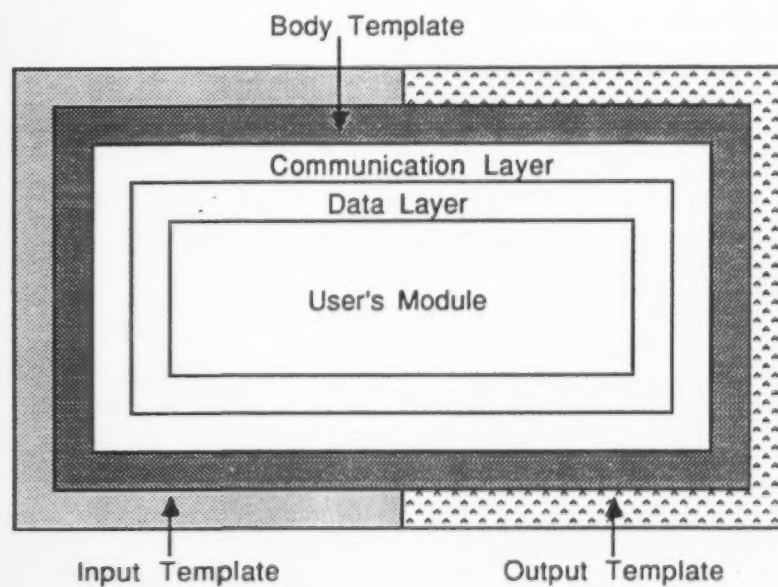


Figure 3.4. Transformation of User Module into Process Code

services remote procedure calls to its entry procedure. Modules are terminated by the system when they are waiting for a remote call, and it is not possible for any of them to receive any more calls. Several other characteristics that regulate the degree of parallelism at execution time and describe the precise behavior of each module are assigned through an action called template attachment. These templates are described in the next subsection.

3.2.2. Templates

Apart from *call* or *reply* statements, a module has no other information regarding synchronization, scheduling, and communication. This information is added through a separate set of attribute bindings known as *template attachments*. Every module needs up to three templates (input, output, and body templates) to fully describe its behavior in a concurrent environment. As described below, an input template is responsible for correct scheduling and synchronization of incoming messages. Similarly, an output template deals with scheduling and synchronization of calls originating from the given module to other modules. The use of a body template is optional. It is used to assign additional characteristics to the modules that modify the module's execution behavior in the distributed environment.

Templates are used to describe the interaction of a module with other modules. Essentially, most distributed programs take advantage of several frequently occurring communication structures, such as pipelines, or master/slave relationships [18,78]. By specifying which template(s) apply to the user's module, *FrameWorks* inserts the necessary code to set up the desired relationship. For example, if a user wants to set up a pipeline of processes, then the user need only specify that the process gets its input from a pipeline, and its output goes to a pipeline.

The Figure 3.4 shows all the augmentations that are made to a user's module by the system to allow proper communication, synchronization and scheduling to happen. In addition to the code for the templates, there are two other layers of software that are

required for the low-level communication package used (similar to [9]). The data layer is concerned with constructing program code necessary for sending and receiving frames in the form acceptable to low-level communication routines. The communication layer is concerned with the selection and routing of messages to the appropriate instance of a process in case there are multiple copies of the module under execution. To distinguish the code in a user's module from its augmented version, the later is referred to as a process.

3.2.3. Input Templates

Currently three types of input templates are supported. These are *initial*, *in_pipeline*, and *assimilator*. Figure 3.5a shows the icons that are used to represent these templates.

3.2.3.1. Initial

A process with initial input template does not receive any input from any other process in the application. Therefore, such a process functions only as a client to other processes and does not serve any remote procedure calls from any other process. Since such a process can not be called by any other process, only the main module of the application may use this template. For example, an application where the *main* module does not have an *entry procedure* would require an initial template.

3.2.3.2. In_pipeline

A process with an *in_pipeline* input template can act as a server to any of its input processes. In this case, the input data for the module's entry procedure is supplied via an input frame by any one of its input processes. Calls from processes are accepted in a first-come first-served manner. In the traditional sense of RPC, this template represents the manner in which a remote procedure accepts its input. For example, this template would be used to specify the input interface for a process that is part of a pipeline.

3.2.3.3. Assimilator

The assimilator type template, however, represents an input interface that is not possible using the conventional remote procedure protocol. In this case, the complete input required by the process is divided into several frames and each input process supplies exactly one of the required frames. This template becomes handy if, for example, a process is to merge the outputs of several processes.

3.2.4. Output Templates

There are three types of output templates as well. Figure 3.5b shows their icons.

3.2.4.1. Out_pipeline

Once again out_pipeline represents the conventional manner in which the called procedure is selected in a RPC protocol. In the case of an out_pipeline output template, there is exactly one instance of each of the processes connected to its output.

3.2.4.2. Manager

Often some processes require more processing time than others and, to keep them in step with other parts of the application, a user may want multiple instances of the same process running. The manager template is concerned with management and scheduling of multiple instances of the same module. The user's module is unaware of the existence of multiple instances of the called module. The manager will select the currently available instance for placing its call. For example, managers can be used to achieve master-slave relationships. The user can specify a fixed number of processes of each kind that are connected to the given process' output. The only restriction that is applicable in this case is that the modules whose multiple instances are run must have in_pipeline as their input template and should have their input connected to the manager.

3.2.4.3. Terminal

This is simply a special case of the `out_pipeline` template. A process with a terminal type output template does not call any other process. Normally, by the time processing of data is completed by a process with a terminal output template, any useful information is already stored in files or tables, or passed back to the calling client process.

Together, input and output templates provide several ways of structuring the interface of a module in a distributed environment. To illustrate the expansion of a user module, consider the module X in Example 1 of Figure 3.5d. Here, the input and output of module X has been structured as `in_pipeline` and `out_pipeline` respectively. Figure 3.6a shows a *FrameWorks* module that is part of a pipeline. Figure 3.6b shows the pseudo code for a possible expansion of the source code in Figure 3.6a. It should be noted how little programming was required in 3.6a to achieve 3.6b. Of course, the savings are much more significant for more complicated structures such as managers or contractors (described below).

3.2.5. Body Templates

Body templates (Figure 3.5c) can provide further choice for specifying the behavior of a module. There are currently two choices for the body template: *executive*, and *contractor*.

3.2.5.1. Executive

An executive template is meant to serve as a user interface to the application and hence only the main module of an application is allowed to use it. The executive template causes the process to have its input, output and error streams directed to the user's terminal. If there is no executive type process in an application, then the complete application can be run in the background without user intervention.

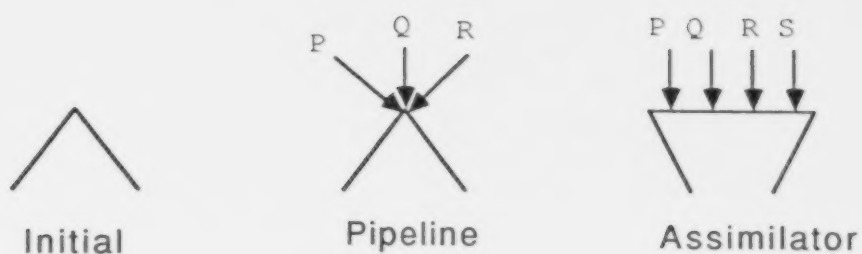


Figure 3.5a. Input Templates

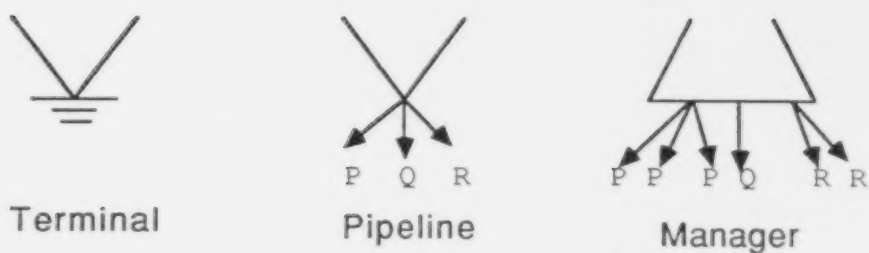


Figure 3.5b. Output Templates

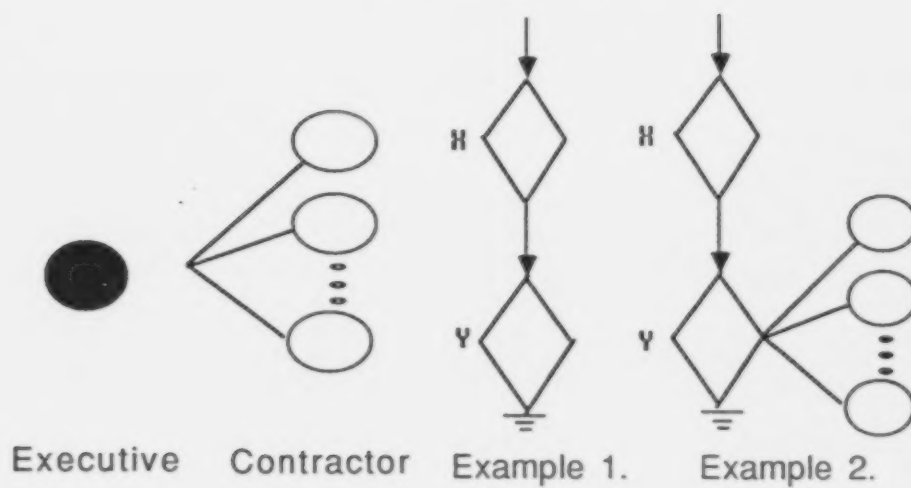


Figure 3.5c. Body Templates

Figure 3.5d. Examples

Figure 3.5. Representation and Use of Templates

```

-----
/* The code in the user module does not have any information regarding */
/* the type of templates that are used to encapsulate it. */

interface_routine( input_frame )
{
    ...
    routine code
    ...
    output = call Y( input );
    ...
}

local_procedures ... /* local procedures may issue remote calls as well */
-----

```

Figure 3.6a. User Module

```

-----
/* The expansion of user module as shown below is based on the */
/* assumption that the module is assigned in_pipeline, out_pipeline, */
/* input and output templates respectively. No body template is used. */

in_pipeline()
{
    while( true ) {
        who = CheckMessage();
        if( who == A_VALID_CLIENT ) {
            ReceiveMessage( who, input );
            interface_routine( input );
        }
        else if( who == FW_MONITOR ) {
            ReceiveMessage( who, message );
            if( message == REQUEST_FOR_STATUS )
                SendMessage( MONITOR, status );
            else if( message == TERMINATION_NOTICE ) {
                CloseBooks();
                Terminate();
            }
            else if {
                /* some other actions (not included) */
            }
            else ErrorHandler();
        }
        else ErrorHandler();
    }
}

interface_routine(input)
{
    ...
    routine code
    ...
    /* expansion of "output=call Y( input )" for out_pipeline */
    /* output template. */
    CheckValidServer( Y );
    status = WaitForStatus( Y );
    if (status != IDLE)
        ErrorHandler();
    SendMessage( Y, input, REPLY_MODE );
    ReceiveMessage( Y, output );
    /* end of expansion */
    ...
}

local_procedures...
-----

```

Figure 3.6b. Expansion of User Module into Process Code

3.2.5.2. Contractor

The default in *FrameWorks* is for a process to serve only one call at a time. A frequently occurring problem arises when some processes in an application require significantly more computation than others, perhaps creating a bottleneck. To solve this problem, one might try partitioning the work done by such a process by breaking it into smaller modules, or try using a manager template that hierarchically distributes the work done by the process. However, each of these structures is defined at design time and requires a fixed number of processors. To increase flexibility at execution time in using idle processors and relieving the designer from specifying the exact number of processors required by such processes, the model provides a body template type called *contractor*. When a module's body is declared as a contractor, it means that the process gets its work done by *employee* processes. An employee process is nothing but a replica of a user's module in execution which has been suitably extended to be able to execute under the control of another coordinating process, called the contractor. Contractors hire and fire employees dynamically at execution time depending upon the workload. Example 2 of Figure 3.5d shows the revised representation of the process Y in Example 1 after it is declared to function as a contractor. *This is done without any change in the source code of the modules.*

3.2.6. Sources of Parallelism

There are two types of parallelism that can be exploited via *FrameWorks*- parallelism due to partitioning, and parallelism due to replication. In *FrameWorks*, unlike the RPC model [9], modules can also be called in a non-blocking manner. Therefore, all processes that have at least one set of input data ready can execute simultaneously. For example, in an application a process A can pass its processed data to another process B via a non-blocking call, and then resume its own processing of the next set of input data.

Contractors and managers provide the means for exploiting parallelism due to

replication. One necessary condition for correct results in this case is that each of the replicated process should do independent computation. That is, the module to be replicated should not contain any state information within itself. The manager represents a hierarchical distribution of data where the output of one process is distributed amongst a number of processes that execute in parallel. The number of processes amongst which the data is divided is specified at application design time. Contractor templates, on the other hand, dynamically vary the number of replicas that may be executing at any time. Depending upon the number of available processors and the processor time requirements of the module, a contractor hires a dynamically varying number of employees to get the job done. The designer of the application does not take any part in the hiring and firing of employees other than simply specifying that the given module should function as a contractor. The actual process of hiring employee processes is managed by the run-time environment and is transparent to the designer. Although a manager provides only a restricted form of services provided by a contractor, it does serve a useful purpose. Since a manager executes only a fixed number of processes, its execution time overhead is expected to be lower than that of a contractor.

3.2.7. Processor Binding in FrameWorks

So far, the problem of binding of processors to various nodes of an application graph has not been discussed. In fact, the issue of processor binding is entirely decoupled from the issue of specification of distributed structuring and development of the application code. This allows the issue of processor selection and allocation to nodes to be handled in accordance with the constraints and features of the underlying architecture. For example, processors can be assigned to various nodes of the graph completely or partially at the time of application design, or it can be left entirely up to the run-time environment. Also, depending upon the implementation, the bindings may be static, lasting throughout the execution of an application, or it may be dynamic, whereby different processors may be bound to a node during an execution. The

FrameWorks model puts no restrictions on how processors should be bound to different nodes. In Chapter 5, a simple notation for binding workstations to the nodes of an application graph is described. Using this notation, a user can specify the assignment of processors completely, partially, or one can leave it entirely up to the system.

3.3. An Example

In this section, a simple example is presented, which is aimed at illustrating the basic features of the *FrameWorks* model and the process of development of a distributed application using the model. This example has been implemented using the *FrameWorks* system that is described in Chapter 5. Descriptions of several other examples that have been implemented are given in Appendix B.

The example under consideration has been taken from graphics and deals with a recent method of computing global illumination for the purpose of generating realistic pictures using computers, called radiosity [47]. The procedure, derived from methods used in thermal engineering, has a fundamental energy equilibrium basis, and correctly models the effect of diffuse light area sources as well as the "color bleeding" effects that are caused by diffuse reflections.

In spite of its merits, the radiosity method is very computation intensive. The application under discussion uses the hemi-cube method of computing radiosity [26]. In our implementation of the above method, the computation of the radiosity for a single environment took about 48 hours of CPU time on a Sun3 workstation [117]. Therefore, it was decided to implement it in a distributed manner.

In the distributed implementation, the complete application is divided into three modules (Figure 3.7a). The module *input* inputs the geometry of the image for which radiosity solutions are to be generated. The complete image is then divided into patches. *Input* selects the patch for which computations are to be done and sends the identification of the patch to the *formfactor* module. *Formfactor* constructs a cube with the given patch at the center of it. The half of the cube that is above the patch is

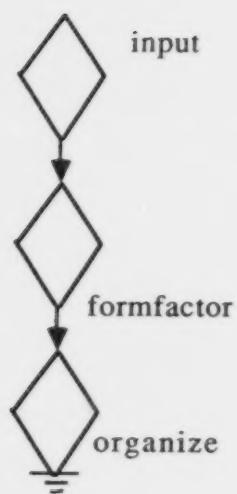


Figure 3.7a

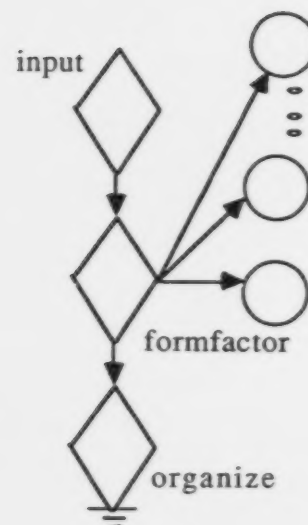


Figure 3.7b

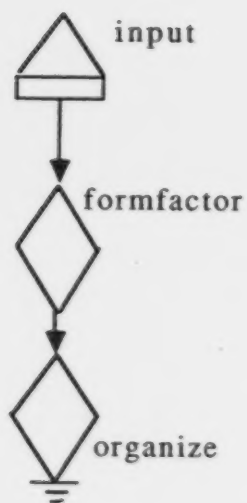


Figure 3.7c

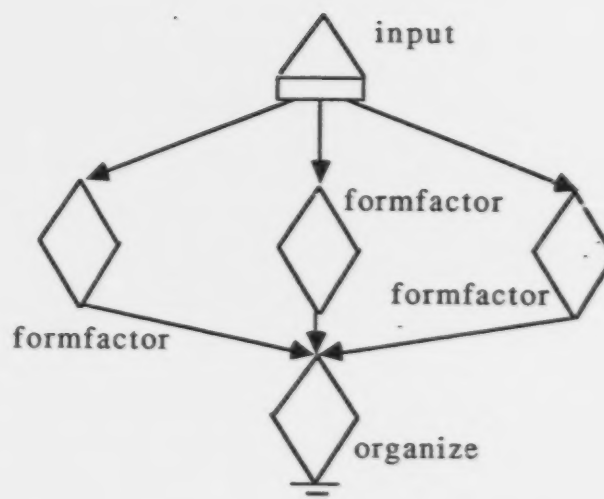


Figure 3.7d

Figure 3.7. Distributed Radiosity

divided into a mesh of pixels and the computation is done for each pixel. The computed results for each patch are sent to the third module called *organize*. *Organize* receives results from the *formfactor* module, does some necessary data conversions, and stores the data into files. Finally, all this data is used for the calculation of radiosity values.

Obviously, this example exploits parallelism due to partitioning of the complete application into three modules which are made to execute in parallel through the use of non-blocking calls. However, the amount of parallelism exploited in this way is small because the computation time requirements of *formfactor* are much higher than those of *input* and *organization*. Therefore, it acts as a bottleneck for the complete application. To remedy the situation, *formfactor* can be assigned a *contractor* body template (Figure 3.7b). Now as many processors as are available are used to speed up the computation. Also, the load balancing in *formfactor* is done dynamically. If there are idle employee processes, they are released. If work is coming in at a faster rate then more employees can be hired. Third, unnecessary context switching is absent. A new employee process is created only if a new (idle) processor is available. If the user decides to run a fixed number of replications of *formfactor*, it can be done by using the configuration shown in Figure 3.7c. In this case, the output template of *input* is constructed as a *manager* which manages the distribution of work load among the specified number of copies of the *formfactor* module. For example, if in Figure 3.7c three copies of *formfactor* are specified to be run, the structure that is actually used at execution time looks as shown in Figure 3.7d. By using a fixed number of replications, the manager template has the advantage of incurring smaller run-time overheads as compared to the *contractor* template.

It should be noted that all the configurations shown in Figure 3.7 have been created simply by changing the application graph. The source code of the modules remains the same. Also, by simply changing the remote procedure call statements to local procedure calls, the complete application can be compiled to run sequentially.

Finally, the configurations such as shown in Figures 3.7b and 3.7c have fairly complicated structures operating behind them. Writing such an application using low-level primitives such as sockets [71] would require considerable effort on the programmer's part. Since such an effort would be replicated, perhaps unwittingly, by other programmers while writing different applications, *FrameWorks* provides an easy way of avoiding this wasted effort.

3.4. Choice of Nomenclature in *FrameWorks*

The names of various templates, such as manager, contractor, and in_pipeline, in *FrameWorks* resemble structures commonly found in everyday life. This choice of nomenclature is by design rather than a random choice, and is based on a methodology known as anthropomorphic programming. Although the use of anthropomorphic metaphors in studies related to computing science is much older, the first definition of this methodology was given by Booth, Schaeffer, and Gentleman [14]. According to their definition anthropomorphic programming is a methodology for structuring programs. By making analogies between processes and people, between concurrency and similar cooperative tasks in everyday life, the programmer can draw solutions that mimic structures evolved by society. The result is often a simple, but elegant solution. It should be noted that such attributions are already an active part of almost every computer scientist's vocabulary. For example, in data processing trees have son, sibling, and father nodes, and queues have heads and tails. Our explanations of various activities is often based on such analogies, e.g., "My process died" or "FTP does not want to print my file". Such illustrations allows us to predict the further behavior of the system. People use them for clarity and understanding.

Examples of the use of anthropomorphic metaphors are not rare. One of the earliest instances of its use can be found in Dijkstra's work, where he explains his solution to the famous dining philosophers problem in the following words [38]:

Instead of N sequential processes cooperating in critical sections via common variables, we take out critical sections and combine them into a $N+1$ st

process, called a "secretary"; the remaining N processes are called "directors". Instead of N equivalent processes, we now have N directors served by a common secretary... What used to be critical sections in the N processes are the director's "call upon the secretary".

The use of analogy is more fundamental here. The vital role a good analogy can play in understanding a problem and formulating its solution is widely acknowledged. In the words of Polya [82]:

Analogy pervades all our thinking, our everyday speech, and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements.

According to Copi [31] "Most of our everyday inferences are by analogy"

Gick and Holyoak say [46] "Analogy pervades thought".

Anthropomorphic programming is not without its share of critics and criticism. Dijkstra who put forward the "directors" and "secretary" solution to the Dining Philosopher Problem later on termed the use of such terminology as "professional immaturity" [39]. We do not know of the reasons for this criticism because Dijkstra gives no further elaboration of his criticism. Also, there is some opposition to the use of this kind of terminology for introducing new users to computers [93].

In general, the problem with analogical models is that they provide only a partial mapping to the computer systems. Also, there are always aspects of the models that are irrelevant to the computer system being modeled. Although experienced users can easily put aside irrelevant aspects, they could be confusing for beginners.

The above criticism is taken as a word of advice while designing our system. There are a few points about our model that should be noted here. First, the *FrameWorks* model is not meant for users that are new to computers. Second, analogies are not treated as substitutes for formal definitions. In the *FrameWorks* model, precise technical descriptions of the structures and functionalities of the templates as well as the constraints that apply to them are provided. By introducing well defined process abstractions, such as managers, and employees, one can think and program anthropomorphically without being confused by the irrelevant details of analogy.

3.5. Critical Review of FrameWorks

The *FrameWorks* model provides a new approach to developing distributed applications in which parallel operations are encoded via templates. A detailed comparison of *FrameWorks* with other similar models and systems for parallel programming will be deferred to chapter 7, until we have described the model as well as the system. At this point, however, a short critical review of the model is presented that summarizes the model's major strengths and weaknesses.

One of the major strengths of the *FrameWorks* model is its ability to separate the specification of the structure of the application from its code. Although this can be achieved in several ways, the concept of template attachment and the use of graphical icons for representing these templates is expected to be a method that is simple to use and understand even by the nonexperts of distributed or parallel programming. The decoupling of templates from modules renders several interesting properties to the model including the ease with which applications can be restructured and converted for sequential/parallel execution. The model also separates the issue of processor binding from the issues of distributed structuring and the development of application code. Therefore, depending upon the constraints of the underlying architecture and user requirements, a suitable scheme for processor allocation can be added. As an example, in chapter 5 we describe an implementation of the *FrameWorks* system that can provide a user not only a tool for utilizing idle workstations on the network but also a complete model for developing distributed applications. A user can exercise complete, partial, or no control on the process of processor binding. Finally, the graph-based approach of the model makes *FrameWorks* inherently suitable for implementing it as a visual model.

A number of researchers have pointed out that programmers rely heavily on commonly occurring structures [18,78] for structuring their parallel applications. Also, there is a large class of parallel applications where synchronization, communication, and scheduling structures are much more restricted. *FrameWorks* can be put in the

category of specialized models, systems, or languages that aim at providing easy generation of commonly occurring forms of synchronization and concurrency [18,64,67,78]. *FrameWorks* is expected to be a useful model for developing a large class of applications that exploit medium and coarse grain concurrencies via commonly occurring structures such as pipelines, fan-in and fan-out structures, and module replications. However, the approach is limited in several ways:

- 1) For large general purpose programs such as an operating system, or a real time transaction system where a group of processes interact or "converse" in an arbitrary manner [44], *FrameWorks* may not be a suitable tool.
- 2) Although the contractor/employee combination involves dynamic scheduling and termination of processes, for simplicity *FrameWorks* does not support dynamic initiation of processes by the user. Therefore, it is more suitable for environments where the computational structure of an application can be specified at compile time. For more complex situations where the pattern of computation is unknown at compile time, languages such as Ada or other low-level tools [23,71] might be more appropriate.
- 3) Although *FrameWorks* provides a facile way of constructing distributed applications, it hides overhead and communication costs from the user. Consequently, the final design may not be as efficient as one that is custom built by an expert programmer. Customized generation of programs using low-level tools is a time consuming and tedious process even for experienced programmers. Programming with tools, such as sockets, has previously been compared to programming at the physical level. However, it might be a suitable alternative for applications that critically depend on achieving maximum efficiency.

In summary, *FrameWorks* is a special purpose tool for the generation of applications that can be structured in terms of commonly occurring communication or synchronization patterns. Such applications can be easily, quickly and correctly generated using *FrameWorks* and they can take advantage of available workstations in a flexible

manner. However, the approach may not be suitable for programs with arbitrarily complicated communication/synchronization structures or programs that aim to attain very high levels of efficiency.

3.6. Conclusions

In this chapter, an outline of the *FrameWorks* model was presented. The goals of *FrameWorks* are: (1) to develop a model suitable for developing distributed applications in a workstation environment, (2) to relieve the programmer from writing low-level error free code for synchronization/communication with other modules, and (3) to develop a software tool that provides the programmer with different levels of control over the use of processors in a workstation environment. The underlying model is expected to make it easy to develop and restructure distributed applications, and to match them to the available resources. In the following chapter, we present formal specifications of the model, and describe an abstract implementation model in a architecture independent manner. Implementation of a visual system based on the *FrameWorks* model for a workstation environment is described in chapter 5.

Chapter 4

FrameWorks: Formal Specification and Implementation Models

4.1. Introduction

In the previous chapter, the basic features of the *FrameWorks* approach for developing parallel applications were described. As the name implies, this chapter is concerned with the formal specification of the *FrameWorks* model. The broad meaning of the term specification is *any information that helps describe the entity being specified*. However, the term specification is used here in a narrower sense. In our terms, we would like to state all the structural and operational requirements that a component should satisfy. The purpose of the specification is three-fold. First, such a specification is intended to state the structure and operation of a *FrameWorks* application. Second, this description can be used to develop abstract implementation models that are independent of any particular hardware or software environment. These implementation models can further be examined for their relative merits and demerits. Third, there are a number of issues related to parallel or distributed systems, such as deadlock and termination, that can be examined using such abstract specifications.

This chapter specifies the structure of a *FrameWorks* application using a set-theoretic notation. The technique used for specifying the behavior of a *FrameWorks* process is a hybrid method based on transition diagrams and programming languages. The technique is called the *General Transition Model* [12,13]. It has been chosen here because of its similarity with the well known state transition diagrams, the ease of understanding that it provides, and its adequacy for our purpose in this chapter.

The rest of the chapter is organized as follows. The following section specifies the structure of a *FrameWorks* application graph. Section 4.3 gives a brief overview of the method used for specifying the operation of a process in the *FrameWorks* model. The behavior of a process is determined by the characteristics of different templates in the model, as well as the characteristics of the *call* and *reply* primitives. This

behavior is specified in section 4.4. These specifications are used in section 4.5 to describe two different implementation models and their communication protocols. In section 4.6, concurrency issues, such as application termination and fairness, are considered for each of the implementation models. Finally, section 4.7 compares the two implementation models and presents some concluding remarks.

4.2. The Structure of FrameWorks Applications

In chapter 3, the structure of a *FrameWorks* application was informally described. In this chapter, we extend the description through a more precise and formal specification of this structure. A *FrameWorks* application can be represented by a directed graph $G = (M, L)$ where M is a finite non-empty set of *nodes*, and L is a set, possibly empty, of directed pairs of nodes called *links*. Each node of the graph is assigned a unique program module m . Exactly one of these program modules contains the *main procedure*. This module is called the *main module*. The node that is assigned the main module is called the *main node*. In the following discussion, no distinction is made between a node and the program module that is assigned to it.

Each node of G is assigned a 3-tuple (t_i, t_o, t_b) where t_i is the input template, t_o is the output template, and t_b is the body template. Also, $t_i \in T_i$, $t_o \in T_o$, and $t_b \in T_b$ where T_i , T_o , and T_b are the sets of available input, output and body templates respectively. Currently, in *FrameWorks*

$$T_i = \{ \text{initial}, \text{in\_pipeline}, \text{assimilator} \},$$

$$T_o = \{ \text{out\_pipeline}, \text{manager}, \text{terminal} \}, \text{ and}$$

$$T_b = \{ \text{empty}, \text{contractor}, \text{executive} \},$$

If a module is not explicitly assigned any body template, it is, by default, assigned the *empty* body template. An *empty* body template does not alter any characteristics of the module it is assigned to. Rather, it is defined here simply to make the concept of template attachment uniform with respect to each type of template.

The expressions $\text{type}_{\text{input}}(m)$, $\text{type}_{\text{output}}(m)$, and $\text{type}_{\text{body}}(m)$ denote the input, out-

put, and body templates respectively of a node m . It is assumed that there exists a boolean function *is_main* such that *is_main*(m) is *true* if the node m is the *main* node, otherwise it is *false*. The directed link between two modules m and n is denoted by $l_{m,n}$. Since, L represents all the links of G , the link between m and n exists if and only if $l_{m,n} \in L$.

4.2.1. Criterion for Replication

As discussed in section 3.2 of the previous chapter, the source code in each module is augmented to provide code for each template as well as to provide data transformation and proper communication among modules. This augmented version of a module is used at execution time. To distinguish the user supplied code from its augmented version, the latter is called a *process*. A process corresponding to a module m is denoted as P_m . During execution, depending upon the templates of a node, single or multiple copies of a process may be present. The concurrently executing processes corresponding to a module are called *replicas* of a module. The set representing all the replicas corresponding to a module m is denoted by C_m . The modules connected to the replicated module are unaware of the existence of multiple replicas. The task of scheduling the replicas of a module and their communication with other nodes is managed by the system in a manner transparent to the programmer. However, in the case where multiple replicas of a module are executed, a programmer must ensure that each of the replicas computes independently and does not leave behind any state information within the process code. Also, access to any external resource must be explicitly coordinated in order to ensure consistency and integrity of the accessed resource. From the structural point of view, in order to be replicated, a module, m , must satisfy one of the following two criterion:

- 1) There must exist another node n such that $l_{n,m} \in L$ and $type_{output}(n) = manager$. That is the input of the replicated node must be connected to a process with *manager* type output template.

- 2) The replicated node must have a *contractor* type body template, i.e.,
 $type_{body}(m) = contractor.$

4.2.2. Frames

Between any two modules, communication takes place in terms of exchanges of user specified data structures called *frames*. A *frame* is a data structure consisting of a sequence of named member data structures. Since *FrameWorks*, like Linda [20], is a model that is embedded in a host programming language, the exact definition of the member data structures is based upon the data types of the host language. The implementation described in this thesis is based on the C language. In this implementation, a frame is defined as an ordinary C structure except that *pointer* data types are not allowed.

Two frames f_1 and f_2 are said to be of the same type (denoted by $f_1 \equiv f_2$) if they consist of the same number of member data structures and the corresponding member data structures in the two frames are of the same type.

4.2.3. Links

As mentioned above, $l_{m,n}$ denotes a directed link from a node m to node n . The link between two nodes is unique which means that there cannot be more than one link between any two modules in an application graph. With each link, $l_{m,n}$, the programmer associates an input frame f_i , and an optional reply frame f_r . The input and reply frames of a link $l_{m,n}$ are denoted by $input(l_{m,n})$ and $reply(l_{m,n})$ respectively.

4.2.4. Criterion for Call

The formal parameter list of an entry procedure is usually of the form

(*para_frame*)

Data between two modules is communicated when one module, m , makes an explicit call to another module, n , by executing one of the following two types of statements

within its source code:

1) call n ($input\_frame$);

2) $reply\_frame = call\ n\ (input\_frame)$;

The first form is called a non-blocking call, whereas the second is called a blocking call. In the case of a blocking call statement, the called module n must eventually reply by executing a reply statement:

reply (rep_frame);

At execution time, the augmented versions of modules communicate via using blocking or non-blocking calls. The non-blocking call "call n ($input\_frame$);" results in the transfer of the data in the $input\_frame$ from a process p to a process q if and only if the following conditions hold:

1) $p \in C_m$,

2) $q \in C_n$, and

3) $l_{m,n} \in L$

These conditions simply state that processes p and q belong to the sets of replicas corresponding to the calling module m and the called module n respectively, and a directed link between m and n exists. The data in $input\_frame$ is used as the value of the formal parameter $para\_frame$ in q . In addition to the above conditions, the types of the communicated frame, frames associated with the link, and the frame declared in the entry procedure's parameter list must match. That is,

4a) $input\_frame \equiv input(l_{m,n}) \equiv para\_frame$

For a blocking call, the types of the replied frame, the reply frame associated with the link, and the reply frame expected by the calling module must match. That is,

4b) $rep\_frame \equiv reply(l_{m,n}) \equiv reply\_frame$

The formal parameter list of an entry procedure may alternatively consist of more than one frame such as

(*para_frame*₁, *para_frame*₂ ... *para_frame*_{*n*})

In this case, the given module must have assimilator as its input template. Also, it is useful here to assume an ordering on the input links $l_1, l_2, \dots, l_n$ of the given node. If the entry procedure of the called node contains n input parameters, there must be exactly n incident links to this module - each supplying a single parameter. In this case, the module can only be called via a non-blocking call, and the i_{th} formal parameter of a node's entry procedure is transferred via the link l_i $0 < i \leq n$. Such an ordering can either be explicitly assigned by the programmer, or it can be implicitly assumed by the order in which these links are created during the process of construction of the application graph.

4.2.5. Structural Constraints on an Application Graphs

There are certain constraints on the structure of correct application graphs. They are either imposed to keep the model simple, or may be placed because of semantic reasons. The application graphs in *FrameWorks* must observe the following constraints:

- (1) For a node m , if $is\_main(m) = false$ then $type_{body}(m) \neq executive$, i.e. only the main module can be assigned the *executive* template.
- (2) For a node m , if $is\_main(m) = true$ then $type_{body}(m) \neq contractor$, i.e. the *main* module cannot be assigned the *contractor* body template.
- (3) For two given nodes m and n , if $l_{m,n} \in L$, then $type_{output}(m) \neq terminal$, or $type_{input}(n) \neq initial$. That is, no link in the application graph can originate from a node with *terminal* output template, or end at a node with *initial* input template.
- (4) For two nodes m and n such that $type_{output}(m) = manager$ and $l_{m,n} \in L$, then there exists no other node p such that $l_{p,n} \in L$. Also, $type_{input}(n) = in\_pipeline$, $is\_main(n) = false$, and $type_{body}(n) = empty$. These constraints mean that the nodes linked to the output of a node that has *manager* output template are con-

strained to have only one input link, *in_pipeline* input template, *empty* body template, and such a node cannot be the *main* node of the application.

Of the constraints mentioned above, (1) has been put merely for the reason of keeping the model simple. Constraint (2) prevents replication of the *main* node. Since, the *main* node contains the main procedure, which is generally used for starting up an application and also perhaps for executing certain initialization steps, it should be executed only once. Constraint (3) is based on the definitions of *initial* and *terminal* templates. The reasons for (4) are mixed. It prevents replication of the main module and also helps to keep the structure of the application graph simple.

4.3. The General Transition Model

To specify the operational aspects of our model, we use the general transition model. This model combines aspects of finite state transition diagrams and programming languages. In this subsection, we give a brief overview of this technique. Further details can be found in [12, 13]. In the general transition model, an entity is specified by the set of possible states assumed by the entity, and the set of possible state transitions. State transitions are assumed to exclude each other in time. The possible states are specified by two components:

- a) a finite state transition diagram, and
- b) a set of program variables which each may assume certain values

The state of an entity is described by:

- a) a token that describes the current active place in the finite state transition diagram, and
- b) the present values of the program variables.

The operation of an entity is defined by the possible state transitions. The operation of a state transition is specified in the form:

```

from <present state>
when <input interaction>
provided <condition>
to <next state>
begin <statement> end;

```

A transition is possible only when the enabling predicate, given by the provided clause, is true. The complete enabling condition required for a transition is the conjunction of from, when, and provided clauses. If the transition is made, the transition function is executed, which is specified by the to and begin clauses. Different clauses of the enabling condition, and the transition function are optional, i.e., they are not required for every transition. The to clause assigns a new state to the entity, and the <statement> part between the begin and end clauses may update the program variables.

The general transition model is a simple yet powerful model for specification of distributed systems. It allows a large system to be partitioned into several smaller sub-components. Each subcomponent can then be described separately. Also, a system can be described at various levels of abstraction. Parallel actions can also be represented. These techniques will be explained if and when used in our description. Details of these appear in [12].

4.4. Dual Nature of FrameWorks Processes

Each process in a *FrameWorks* application can potentially function in the dual role of a client as well as a server. Processes repeatedly serve calls placed on them by processes connected to their input and, therefore, function as servers to other processes. During the period of serving these calls, a process may call other processes connected to its output and thus may become a client. This behavior is common for all *FrameWorks* processes irrespective of their templates. In this section, we describe this abstract behavior of *FrameWorks* processes. This description is used to further explain characteristics of different templates.

4.4.1. FrameWorks Processes as Servers

For sake of clarity, throughout this chapter the enabling predicates and transition functions associated with various transitions of a general transition diagram are organized in a separate table. The input interaction part of a state transition, if present, is implied by the name of the transition. Figure 4.1 shows the transition diagram for a *FrameWorks* process. When started, each process executes some initialization steps which depends upon the templates of the process. For example, a process with *contractor* body template may hire the minimum specified number of employee processes. After the initialization steps, the main module starts the execution of the application by invoking its main procedure (transition *START*). Other modules wait for their required number of input frames to arrive at their input. The arrived input frames are examined (transition *IN_DATA*), and when all the input frames required by a process have arrived, it is indicated by the *true* value of the boolean expression *in_criterion*, called the input criterion. The exact structure of *in_criterion* depends upon the input template, and it is defined later in this section for various input templates of *FrameWorks*. In case more than the required number of client processes are ready with their input frames, a client selection function, executed by the *select(input)* function, decides which call is to be served next (transition *COMPLETE*). Frames from the selected clients are received, and the entry procedure of the module is called (transition *OPERATE*). During the execution of the entry procedure, represented by the state *EI* (Execute and Interact) in Figure 4.1, other processes may be called. Upon completion of the entry procedure, the process waits for further input calls to arrive and this cycle of operation is repeated until the termination condition, given by T_c , is true at which time the process terminates (transition *TERM*).

The operation of the *main* module is slightly different. In this case, the main procedure of the module starts the execution of the module (transition *START*), and during the execution of the main module other processes may be called. However, after the completion of the main procedure (transition *NEXT*), the main module operates exactly

in the same manner as any other module, i.e. it repeatedly serves in-coming calls. In the case that the main module has an *initial* input template, it cannot serve any calls and, therefore after completion of its main procedure, it simply waits for the termination condition to be true.

4.4.1.1. Definition of Input Criterion and Client Selection Function

In addition to the structural constraints specified in section 4.2, input templates are distinguished by the manner in which they receive their input data. These distinctions are defined by the input criterion *in_criterion* and the input selection function *select*. The *in_criterion* defines a logical condition which is used to determine whether an input template would accept input calls, and what input is required to invoke its entry procedure. The *select* function discriminates among the presently arrived input calls and selects the client(s) to be served. For our existing templates, *in_criterion* and *select* are defined as follows:

Let us assume that a module m has $m_1, m_2 \dots m_n$ as its input modules, and $C_{m_1}, C_{m_2}, \dots C_{m_n}$ are the sets of corresponding replicas respectively. Further, we assume the existence of boolean variables $o_1, o_2, \dots o_n$ such that presence of an input frame from an input process $p_i, p_i \in C_{m_i}$, sets the value of variable $o_i, 1 \leq i < n$, to *true*. Now, for our input templates, *in_criterion* is defined as follows:

- a) *in_criterion* : *false* (*initial* template),
- b) *in_criterion* : $o_1 \vee o_2 \vee o_3 \vee \dots \vee o_n$ (*in_pipeline* template),
- c) *in_criterion* : $o_1 \wedge o_2 \wedge o_3 \vee \dots \wedge o_n$ (*assimilator* template),

where " $\vee$ " and " $\wedge$ " denote logical "or" and "and" conditions respectively.

A process can receive input frames from its client processes only after *in_criterion* has turned *true* (transition *COMPLETE*). For the *initial* template, *in_criterion* is assigned *false*, which means that a process with this template cannot accept any incoming calls. For an *in_pipeline* template, input from any one of its input

processes is sufficient whereas for an *assimilator*, an input frame from each of its input processes is required.

The *select* function defines the criterion that is used to select the client(s) to be served next. Currently, for *in_pipeline* as well as *assimilator*, the next client(s) and input frame(s) are selected on the basis of a *first come first served* (FCFS) policy.

4.4.2. FrameWorks Processes as Clients

Figure 4.1 describes the server side of a process. To understand the client side, the state *EI* (Execute and Interact) of Figure 4.1 must be expanded. It is in this state that a process executes its main procedure or entry procedure. During this, the process may execute a *call* statement and thus become a client to some other process. Figure 4.2 gives the expansion of the *EI* state of Figure 4.1. The execution of a *call* statement involves selecting a replica of the named server to which the current call is scheduled (transition *CALL*). This is done by executing a server selection function called *get_id*. This is important because there may be more than one replica of the named module under execution. Execution of a blocking call involves sending the input frame to the called process (transition *BLOCKING*). The execution of the client process is suspended until a reply frame is received (transition *RECEIVE*). For non-blocking calls (transition *NONBLOCKING*), no reply frame is received. The client resumes its own execution as soon as the input frame is delivered (transition *DATA_SENT*).

Apart from executing *call* statements, a module's entry procedure may require execution of *reply* statements. Strictly speaking, during the execution of a *reply* statement, a process functions as a server. However, this interaction is a part of the expansion of the *EI* state, and therefore it is included in Figure 4.2. The execution of the *reply* statement involves sending the reply frame to the client process (transitions *REPLY* and *REPLY_SENT*).

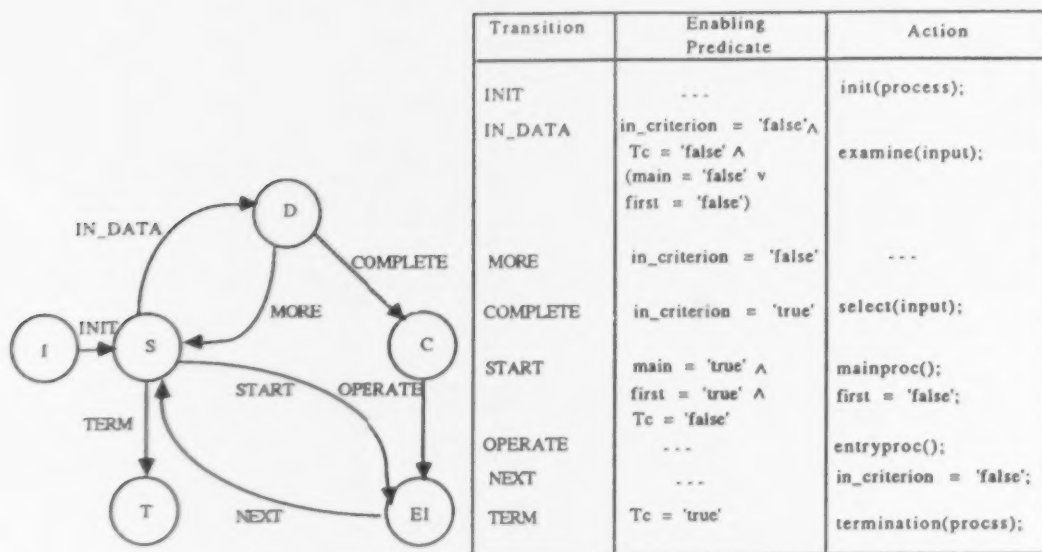


Figure 4.1. FrameWorks Process as a Server

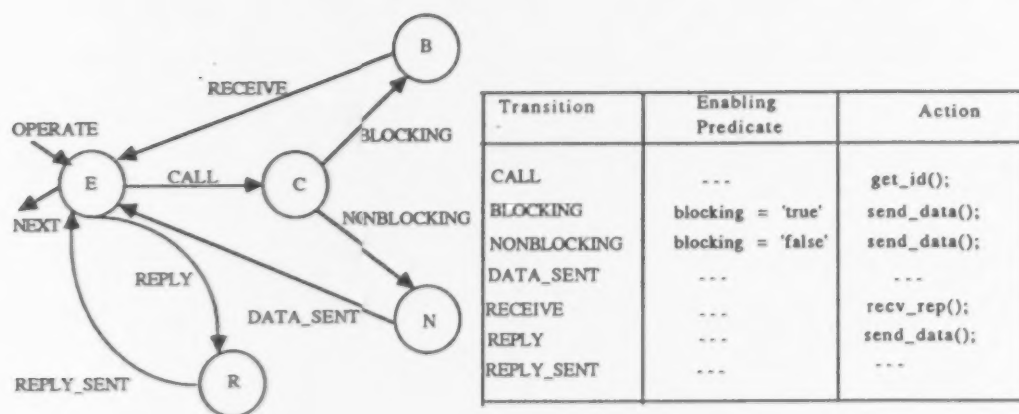


Figure 4.2. Expansion off State EI in Figure 4.1

4.4.2.1. Definition of Server Selection Function

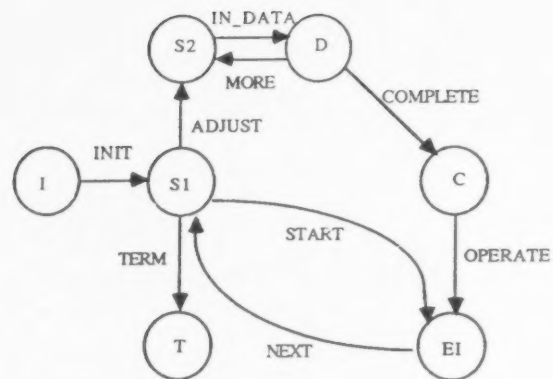
The *get_id* function describes the criterion that is used to select the server process for scheduling a call in case there is more than one replica of the called server. In our current definition, the process that has been waiting the longest is selected for processing of the current call.

4.4.3. Body Templates

The behavior of templates with *executive* or *contractor* body templates can be described within the above mentioned behavior of a process. For a process with the *executive* body template, the input, output, and error streams of the process are appropriately redirected during the transition *INIT* in Figure 4.1. In the case of a *contractor* body template, a different number of replicas of the module may be present at different times during the entire execution period. This can be specified through the expansion of state *S* of Figure 4.1 into states S_1 and S_2 . The expanded state diagram is shown in Figure 4.3. At the beginning of each cycle of servicing the external calls, a contractor process evaluates its criterion for adding a new replica of itself, or terminating an existing replica (transition *ADJUST*). This criterion may be based on a number of factors such as, the average amount of delay in servicing an in-coming call, the number of available processors in the environment, etc. Depending upon the results of the evaluation, the number of replicas that are employed is adjusted. Therefore, if the calls are coming in at a faster rate, more replicas of the process may be created. On the other hand, if several replicas are idle, they may be terminated.

4.5. Implementation Models and Their Communication Protocols

From a user's point of view, the only messages exchanged among processes are the frames associated with *call* and *reply* statements. Such messages are called *user messages*. To ensure proper synchronization and scheduling, however, exchange of additional control information is required. These messages are called *control messages*. Control messages may be directly exchanged between the communicating



Transition	Enabling Predicate	Action
INIT	---	init(process);
ADJUST	---	adjust_emp();
IN_DATA	in_criterion = 'false' ∧ Tc = 'false' ∧ (main = 'false' ∨ first = 'false')	examine(input);
MORE	in_criterion = 'false'	---
COMPLETE	in_criterion = 'true'	select(input);
START	main = 'true' ∧ first = 'true' ∧ Tc = 'false'	mainproc(); first = 'false';
OPERATE	---	entryproc();
NEXT	---	in_criterion = 'false';
TERM	Tc = 'true'	termination(proc);

Figure 4.3. FrameWorks Process as a Client

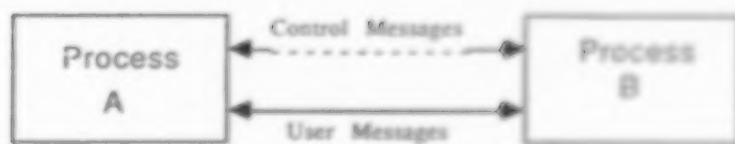


Figure 4.4a. Direct Communication

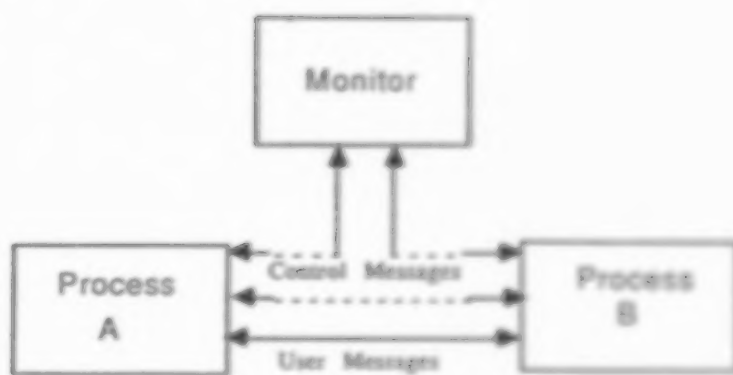


Figure 4.4b. Monitor Assisted Communication

modules (Figure 4.4a), or the exchange might also involve some intermediate system component (Figure 4.4b). The exchange of control messages is transparent to the user of the application, and it is carried out according to some predetermined communication protocols of the underlying implementation model. The design of such models involves serious trade offs among efficiency of the implementation, additional constraints on the application graphs, and global monitoring and services that can be provided to the user.

In this section, two implementation models are presented which sharply differ from each other with respect to these considerations. The first of these is called the *restricted model* because it allows only acyclic application graphs. An advantage of this model is that it requires the exchange of very few control messages and therefore can be implemented efficiently. However, it has serious disadvantages as far as global monitoring and control of an application is concerned. The other model is called the *general model*. It requires the exchange of more control messages than the restricted model. However, it does not impose any additional constraints on the application graph and it can be used to provide a variety of global services to a user. These aspects will be discussed in detail later in this chapter.

4.5.1. The Restricted Model

In addition to the constraints described in section 4.2.5, the restricted model puts the extra constraint on the application graph that there cannot be a directed cycle in the application graph. The communication protocols for the restricted model for a client and a server process are illustrated in Figure 4.5a and Figure 4.5b respectively. For general transition diagrams that describe communication protocols, it is useful to mark the transitions that result in an exchange of messages. In our diagrams, transmission or receipt of a message is indicated by an up arrow (" $\uparrow$ ") or a down arrow (" $\downarrow$ ") respectively.

First, the client's side is considered (Figure 4.5a). Initially, all processes are

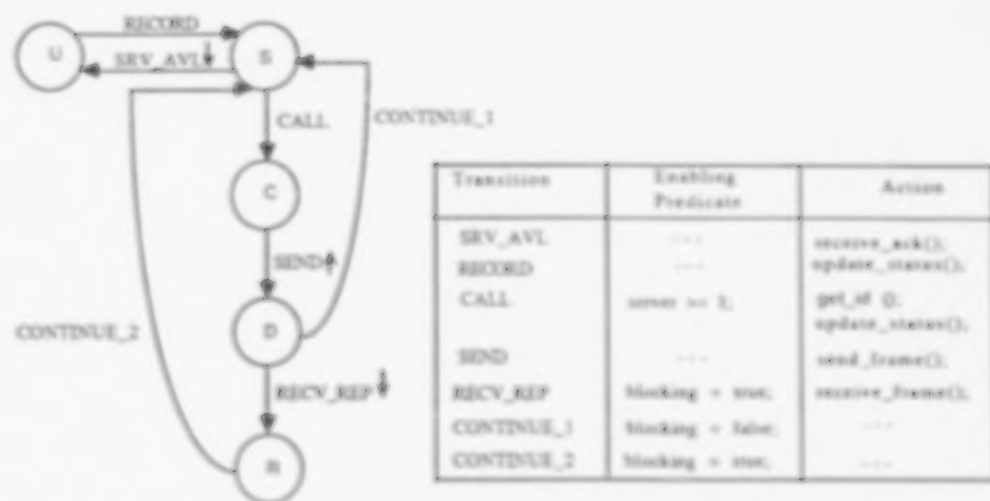


Figure 4.5a. Client Side in the Restricted Model

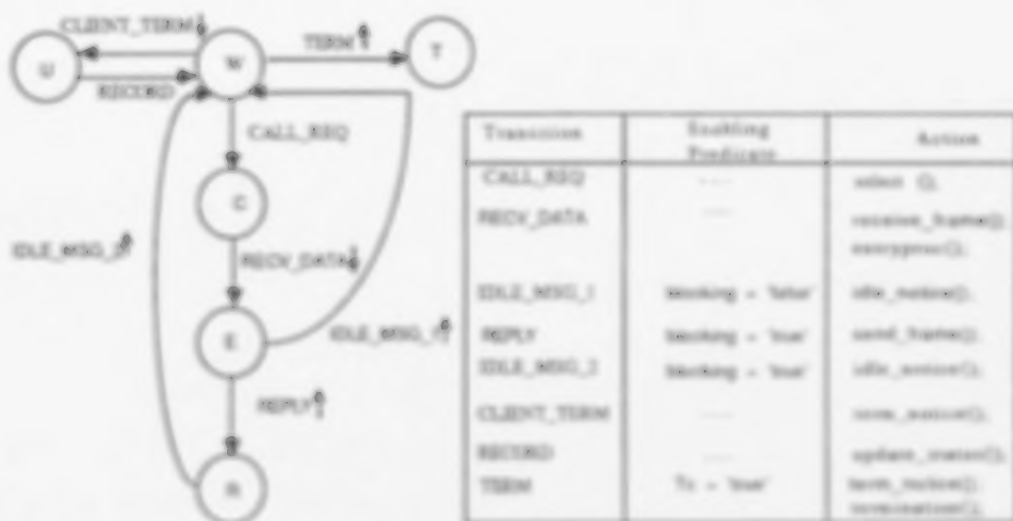


Figure 4.5b. Server Side in the Restricted Model

marked idle. When a *call* statement is encountered, based on the criterion defined in section 4.1, the client selects a server with the specified name from its list of currently available servers (transition *CALL*). If no such server is available, the client is blocked waiting for an idle message to arrive from any server connected to its output (transitions *SRV_AVL* and *RECORD*). When an idle server with the same name is found, a message containing the type of call (blocking or non-blocking) along with the input frame is sent to the server (transition *SEND*). For a non-blocking call, the client returns immediately and resumes its own execution (transition *CONTINUE_1*). For blocking calls, the sending of input frame is followed by receiving the reply frame (transition *RECV_REP*), after which the client resumes its execution (transition *CONTINUE_2*).

On the server side (Figure 4.5b), the process initially waits for a message from any of its clients. If call requests from one or more clients are present, the client to be served next is selected (transition *CALL_REQ*). The input frame(s) from the selected client is received (transition *RECV_DATA*), and the server's entry procedure is called. For a blocking call, a reply frame is sent before the entry procedure returns (transition *REPLY*). On return from its entry procedure, the server sends an idle notice to the client(s) that it just served (transition *IDLE_MSG_1* or *IDLE_MSG_2*).

In addition to the idle notice, the only other type of control message exchanged in the restricted model is the message exchanged for the purpose of terminating an application. This exchange is dealt within the transitions *CLIENT_TERM*, *RECORD*, and *TERM* of Figure 4.5b. The termination procedure for a *FrameWorks* application is discussed in section 4.6 along with other concurrency issues.

4.5.2. The General Model

The restricted model provides an efficient scheme for implementing a restricted version of *FrameWorks*. In addition to limiting the allowed application graphs to the acyclic category, the restricted model suffers from the limitations resulting from the

lack of global information regarding the status of various processes. In a distributed application, such information is vital for such purposes as, debugging, tuning, and visualization. This aspect will become clearer during the discussion in the following sections. In this section, a different model of communication is presented which removes the above-mentioned shortcomings of the restricted model. However, it assumes the existence of a system component called *monitor*. In this model, each user message (*call* or *reply*) requires transmission of a few additional control messages. The information in these control messages allows the monitor to maintain a fairly accurate information on the status of various processes.

The transition diagrams for the client and the server sides are shown in Figure 4.6a and 4.6b respectively. The transition diagram for the monitor process is split into two parts illustrating its interaction with the client side (Figure 4.7a) and the server side (Figure 4.7b). Initially, to call a server, the client process sends the monitor a message containing the name of the desired server along with the type of the call (blocking or non-blocking) that is going to be made (transitions *CALL_REQ* in Figure 4.6a and 4.7a). The monitor maintains the status information for all the processes in the application. The detailed structure of this information is presented in the next sub-section. Now, if a replica of the server being requested is currently idle then the client process is sent the *process_id* of the available server. This is shown by the transition *CLIENT_ID* in Figure 4.6a and the transition *RECV_ID* in Figure 4.7a. Otherwise, the monitor simply rejects this request (transition *NO_SRV* in Figure 4.7a). The client then simply waits till a replica of the desired server becomes available. After sending the *process_id* of the available server to the client, the monitor updates its data structures (transition *RECORD* in Figure 4.7a). If a blocking call is being made, the monitor marks the client process as now waiting for a reply from the server process. The client process, after receiving the *process_id* of the desired server, sends the input frame for processing to the specified process (transition *SEND* in Figure 4.6a). If the call being made is non-blocking, the client immediately resumes its execution at the

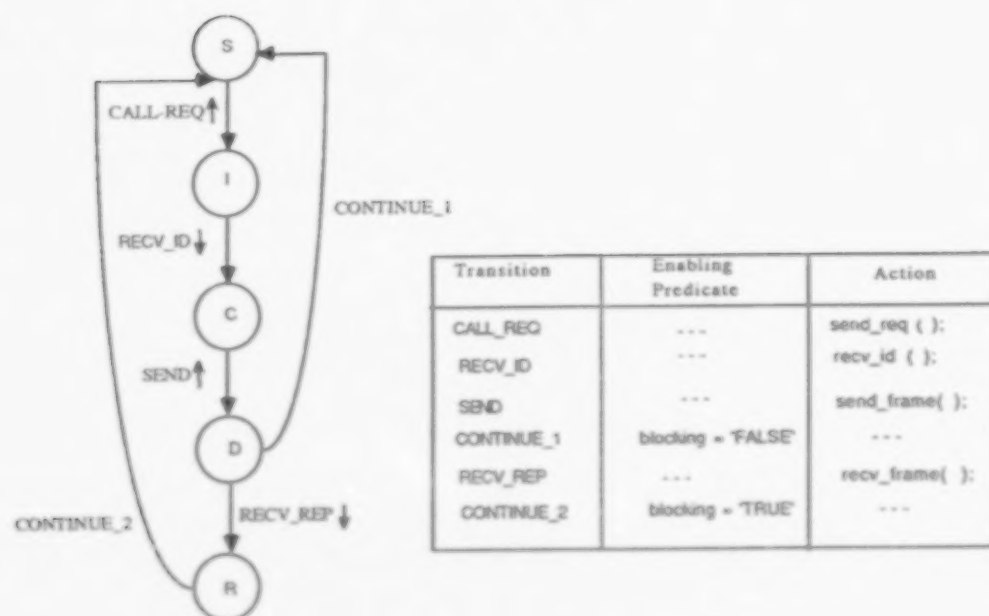


Figure 4.6a. Client Side in the General Model

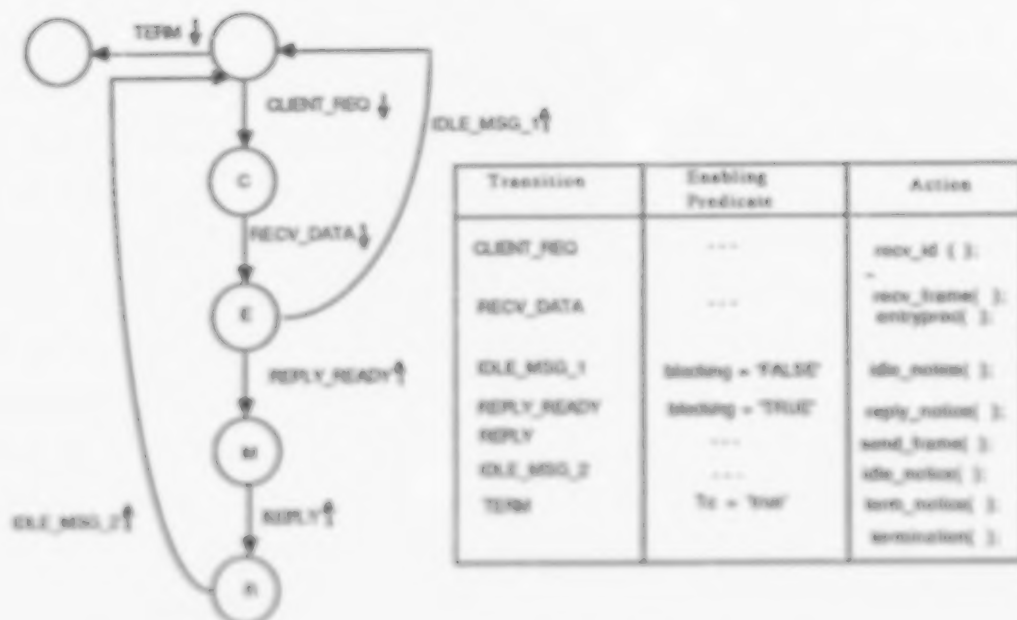


Figure 4.6b. Server Side in the General Model

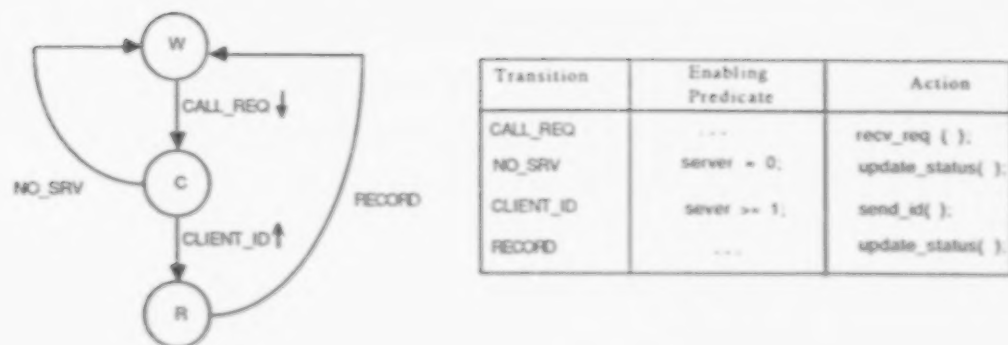


Figure 4.7a. Monitor's Interaction with the Client Side

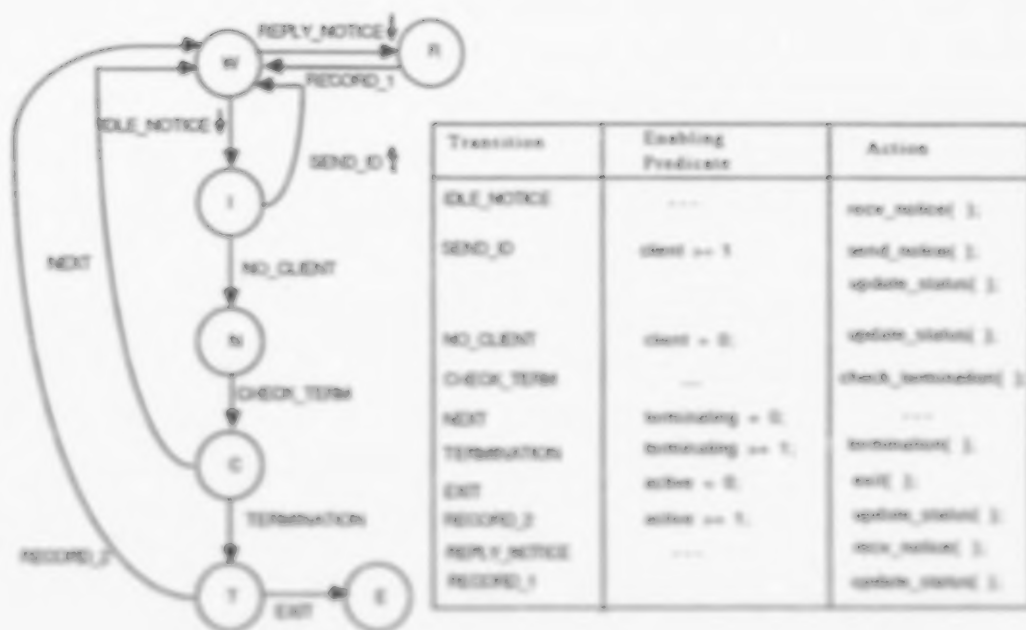


Figure 4.7b. Monitor's Interaction with the Server Side

statement following the *call* statement (transition *CONTINUE_1* in Figure 4.6a). Otherwise, it waits to receive a reply frame from the server process. After receiving the reply frame (transition *RECV_REP* in Figure 4.6a), it resumes its execution (transition *CONTINUE_2* in Figure 4.6a).

When a server becomes idle, it informs the monitor of its readiness to accept calls from clients: This is shown by the transitions *IDLE_MSG_1* or *IDLE_MSG_2* in Figure 4.6b. The monitor receives the idle notice from the server (transition *IDLE_NOTICE* in Figure 4.7b), and examines the state of all its clients. If a client has called the server, then a currently idle replica of the named server is selected by the monitor. The idle server is then sent the *process_id* of the client process along with the type of call being made. This interaction between monitor and a server is shown by the transitions *SEND_ID* in Figure 4.7b and transition *CLIENT_REQ* in Figure 4.6b. On receiving this information, the server process receives the input frame from the client and calls its entry procedure with the received data (transition *RECV_DATA* in Figure 4.6b). If the call being currently served is of blocking type, a reply message is needed to be sent to the client. When the server is ready to reply, it sends the monitor *REPLY_READY* message. The monitor then updates its records to reflect the change in the status of the client (transitions *REPLY_NOTICE* and *RECORD_1* in Figure 4.7b). After this, the server sends the reply frame to the client process (transition *REPLY* in Figure 4.6b). At this point, the server may still be executing the entry procedure. When it returns from the entry procedure, it informs the monitor of its idle state (transitions *IDLE_MSG_1* or *IDLE_MSG_2* in Figure 4.6b). If any process is currently waiting to call the server, the monitor sends the server its client-id and this cycle of operation is repeated.

When a server sends an idle message to monitor (*IDLE_MSG_1* or *IDLE_MSG_2* in Figure 4.6b), there may not be any client waiting for the server. The monitor, in this case, first records the idle status of the server (transition *NO_CLIENT* in Figure 4.7b). Now, it is also possible that this server will never be required by any client. Determination of this condition is a part of the application termination procedure which is

dealt within the following section. The algorithm to determine this is executed (transition *CHECK_TERM*). If no process is ready for termination, the monitor updates its data structures and begins to wait for the next message from any process in the application (transition *NEXT* in Figure 4.7b). If one or more processes are ready for termination, steps to terminate them are taken. This is done within the transition *TERM* in Figure 4.6b and transition *TERMINATION* in Figure 4.7b. The monitor then updates its data structure and waits for the next message from any process in the application (transition *RECORD_2* in Figure 4.7b). Finally, when all processes have terminated, the monitor exits and the application terminates (transition *EXIT* in Figure 4.7b).

4.5.2.1. Activity and TimeStamp Matrices

As described above, the monitor coordinates the interaction among user processes. The monitor can also be used to provide global services to an application such as detection of its termination condition. To accomplish these, we assume the existence of two matrices called *Activity matrix* and *TimeStamp matrix*. If the application contains n processes, the size of the activity matrix is $n \times n$. Each process in the application is assigned a unique logical identification number i ($0 \leq i < n$). The element of the activity matrix in the i_{th} row and j_{th} column (denoted by $A(i, j)$ where $i \neq j$ and $0 \leq i < n$, and $0 \leq j < n$) may assume one of the following values:

- $A(i, j)$
 - $= \epsilon$ which means there is no link from process i to process j and therefore $A(i, j)$ is undefined.
 - $= w_n$ indicating that process i is waiting to place a non-blocking call to process j .
 - $= w_b$ indicating that process i is waiting to place a blocking call to process j .
 - $= w_d$ indicating that process j is waiting for input data from process i .
 - $= w_r$ process i is waiting to receive back a reply from process j .

For $0 \leq i < n$, $A(i, i)$ may assume following values:

- = *ACTIVE* indicating that the process i is active,
- = *IDLE* indicating that the process i is idle and waiting for all the input data to arrive, and
- = *TERMINATED* indicating that the process i has been terminated.

At the beginning of the application the main module is initialized as *ACTIVE*, and all other processes are initialized as *IDLE*. An accompanying matrix, called the TimeStamp matrix, contains timing information by generating *time-stamps* as events occur during the execution of an application. The TimeStamp matrix has the same structure as the activity matrix. An element $T(i, j)$ of the TimeStamp matrix is defined only if the corresponding element $A(i, j)$ in Activity matrix is defined. An element of the TimeStamp matrix that has a defined value contains the local time when the value of the corresponding element $A(i, j)$ was last changed.

4.6. Concurrency Issues

In addition to correctly dealing with the scheduling and synchronization of user messages, an implementation model may also handle concurrency issues such as fairness, application termination, and deadlock. In the following subsections, we introduce these issues and discuss how the implementation models described in the previous section deal with these issues.

4.6.1. Fairness

The definition of fairness in concurrent systems states that if a process is trying to access a resource, it should eventually be able to do so, provided that other processes relinquish control of the resource. This is a desirable property for any synchronization mechanism because it ensures that no process is blocked permanently for a resource. In general, a synchronization mechanism is *fair* if no process is permanently waiting for a condition that occurs infinitely often. It is *bounded fair* if there is an upper limit on how long a process waits for a condition if the condition occurs infinitely often [1].

In *FrameWorks*, the issue of fairness arises whenever clients compete with each other for accessing servers. Assuming that entry procedure of each module always returns after a finite amount of time, the definition of input templates in section 4.4 ensures the bounded fairness of input templates.

First, we examine the enforcement of the fairness criterion in the restricted model. Let us suppose that there are p client processes waiting for a single server. In this case, the client is delayed for at most one allocation of the server process to each of the competing clients.

A similar issue of fairness arises when several replicas of a server compete to serve calls arriving from one of their clients. In the case of the restricted model, clients keep record of the time when its servers fall idle. Therefore, a bounded fair strategy can be implemented. Once again, a server would wait for at most one selection of other servers by each of the clients.

In the general model, the Activity and TimeStamp matrices contain global information regarding the status of each process in an application. Therefore, it is simpler to enforce the criterion of fairness in the general model than in the restricted model. In the case of clients waiting for a server, let us suppose there are p clients waiting for a server with q replicas. In the general model, it is possible to enforce a fairness criterion by which a client would wait for at most $(p - 1)$ allocations to other clients.

In the case where q replicas of a server are waiting to receive calls from p client processes, a particular replica of the server process would wait for at most $(q - 1)$ selections of other replicas of the given server. It should be noted that because of the availability of information regarding the status of all processes in the Activity and TimeStamp matrices, the enforcement of the fairness criterion in the case of the general model is much simpler than in the case of the restricted model.

4.6.2. Termination

Each server in a *FrameWorks* application implements a reactive process which simply reacts to the incoming requests for calls. Such a process cannot terminate itself because it may be required by other processes. However, during the execution of an application, there may be situations when a process can safely terminate because it will no longer be required by any other process. The problem of terminating a process in a distributed program may depend upon some global state of the system or on some local state. The problem of distributed termination is to detect when one or more processes are ready to terminate and to notify them so that they will terminate. Similar problems arise in other models as well. For example, Francez [42] and Cohen and Lehmann [27] studied the problem of termination for the CSP model.

In the restricted model, the problem of termination is handled via a special control message. The process of application termination is initiated by the main modules. When the main module completes execution of the main procedure, it sends notice of its termination to all its output processes. After this, the main process terminates. A process keeps a record of its alive clients (transition *CLIENT_TERM* and *RECORD*). When all the clients of a process have terminated and the process itself is idle, it is ready for termination. It then sends a termination message (transition *TERM* in Figure 4.5b) to all its clients. After this, the process terminates itself. Thus the process of termination starts from the main module and ends at the modules with terminal templates.

In the general model, the information in the Activity matrix can be used to decide whether a single process or a group of processes can be terminated. In this section we present two algorithms to determine the termination procedure for a *FrameWorks* application.

Algorithm_1

```

boolean CheckTermination(A)
Activity_Matrix A(N, N);
{
    int i;
    boolean Terminate;

    Terminate = true;
    for (i=0; i< N; i++) {
        if A(i, i) != IDLE;
            Terminate = false;
        if (Terminate == false)
            break;
    }
    return(Terminate);
}

```

Figure 4.8. Algorithm_1 for Application Termination

Algorithm_1, using a C-like notation, is described in Figure 4.8. The algorithm works as follows: Every time a process changes its state to idle, the algorithm is executed. The CheckTermination procedure determines if all the processes in the application are currently idle. If the algorithm returns *true*, all the processes in the application are ready for termination. In this case, a termination message is sent to every process. On receiving a termination message, a process acknowledges the message and carries out its own termination.

Algorithm_1 is quite simple and computationally inexpensive. It terminates all the processes in the application together. However, it might sometimes be desirable to carry out the termination procedure in a stepwise fashion whereby the monitor detects situations where a process or a group of processes fall idle and they will not receive any further input. Such processes are marked as *ready for Termination*. Such processes can be terminated while processes that are not yet ready for termination may continue with their execution. This algorithm is computationally significantly more expensive than Algorithm_1. The algorithm, called Algorithm_2, works as follows (Figure 4.9):

Algorithm_2

```

Set A_set, L_set, I_set, T_set;

CheckTermination(G, A)
Application_Graph G;
Activity_Matrix A(N, N);
{
    int i, process_x, process_y;

    /* first prepare sets of idle and active processes */
    for(i=0; i<N; i++){
        if(i == IDLE)
            add_set(I_set, i);
        else if (i != TERMINATED)
            add_set(A_set, i);
    }

    for (each process_x ∈ I_set){
        if (live_inputs(process_x) == 0){ /* procedure live_inputs returns the */
                                          /* number of live clients of the */
                                          /* given process */
            add_set(T_set, process_x);
            delete_set(I_set, process_x);
        }
    }

    for (each process_y ∈ A_set){
        accessibility(process_y);
    }

    T_set = (T_set) ∪ (I_set - L_set);
    return(&T_set);
}

accessibility(process)
int process, process_o;
{
    if (is_member(process, L_set)
        return;
    else
        add_set(process, L_set);

    for (each output process process_o of process)
        accessibility(process_o);
}

```

Figure 4.9. Algorithm_2 for Application Termination

A process that has finished processing all of its current input data, and has no immediately pending input data to be processed, is considered idle. The algorithm initially groups all the alive processes of an application into two non-intersecting sets: the

set of processes that are waiting for any input data (I_set), and the set of other alive processes (A_set). The processes in the I_set that do not have any input process are moved to the set of processes that are ready for termination (T_set). Now, the processes in the A_set can call any *IDLE* processes connected to their output, and such activated processes can in turn call *IDLE* processes connected to their output. All the processes in I_set that can possibly be activated in this manner are moved to a new set called L_set . Now, the difference set ($I\_set - L\_set$) gives the set of processes that are idle and also cannot be activated by any active process. Such process are added to the set of processes that are ready for termination (T_set). The procedure returns the address of T_set .

4.6.3. Deadlocks

In a concurrent system a process is said to be in a deadlock if it is waiting for an event that would never occur [61]. A deadlock in a concurrent system can either occur when processes compete amongst each other for resources or when they communicate with each other. The problem of deadlocks has been studied extensively in operating systems [51, 60, 61, 68, 75] as well as database systems [21, 22, 48, 76]. Concurrent programming languages often allow constructs that are prone to deadlocks [114]. For example, in CSP a deadlock can occur when two processes are waiting to send a message to each other at the same time. In concurrent languages, the task of handling deadlocks is more complicated since it may be a result of some logical error in the program and therefore the system is hardly in a position to resolve it. Therefore, most concurrent languages treat deadlocks as a programmer error and leave the responsibility of detecting and correcting it on the programmer.

In the restricted model, although the threat of deadlocks is much reduced because of the acyclic nature of the application graphs, deadlocks may still be present. As an example, Figure 4.10a shows an application graph which has no directed cycles but it still can face a deadlock situation. Figure 4.10b shows a possible deadlock situation in

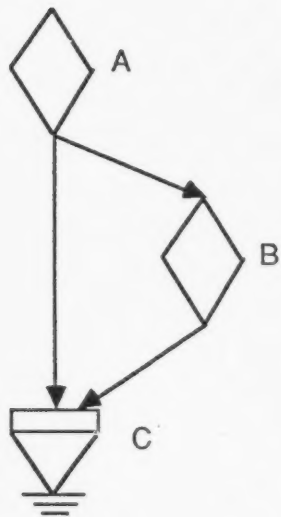


Figure 4.10a

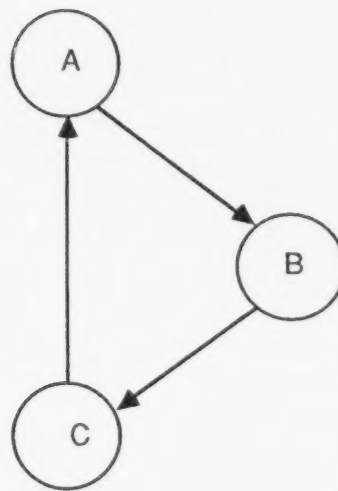


Figure 4.10b

Figure 4.10. An Example of Deadlock in the Restricted Model

this application. In this situation, module A is waiting to call module B, module B is waiting to call module C, while module C is waiting for an input frame from module A. Due to lack of global information in the restricted model, it is not possible to detect deadlock situations.

In the general model, deadlocks can occur when processes call each other in a blocking mode and form a cycle, or several processes are blocked waiting to call each other in a cyclic manner, or a combination of the above two conditions that results in a cyclic wait condition. Algorithms based on well known cycle detection techniques can be developed for execution time detection of deadlocks. A deadlock detection algorithm needs to be executed whenever a process begins to wait for another process. Such algorithms consume significant amounts of time in the monitor and can become bottleneck for the whole application. A simpler way to deal with such conditions is to use the global status information present in the monitor to construct a *wait_for_graph* for the application which can be displayed to the user [21]. Such a treatment is probably more effective and computationally inexpensive. In Chapter 5, we describe a tool to accomplish this purpose.

4.7. Conclusions

The chapter gives formal specification for the structure and operation of a *FrameWorks* application. Also, two abstract models for implementing the *FrameWorks* model are presented. The following chapter describes implementation of two versions of the *FrameWorks* system based on these abstract models.

Chapter 5

The FrameWorks System

5.1. Introduction

The previous two chapters described the *FrameWorks* model for developing parallel applications. So far, the description has been independent of any particular hardware or software environment. To assess the practical feasibility of the model in a workstation environment and to gain some more insights into it, two preliminary implementations of the *FrameWorks* system were done at the Department of Computing Science, University of Alberta. These implementations are based on the restricted and general implementation models described in the previous chapter. Throughout this chapter and during the remainder of this thesis, these two implementations of the model will be referred to as *FrameWorks(R)* and *FrameWorks(G)* systems respectively. This chapter is concerned with the discussion of the architecture of the two systems and preliminary experimental results regarding the performance of the systems. The implementations use a pool of various versions of Sun-3 workstations (Sun-3/50, Sun-3/60, etc.) which are interconnected as a local area network using the Ethernet interfaces. Workstations run under the Sun OS 3.5 operating system and provide basic facilities for remote process creation and message passing. The operating system also supports a distributed file system mechanism called, *Network File System*.

The following section provides an overview of the process of developing distributed applications using the *FrameWorks* system. A description of the components of *FrameWorks* is given in Section 5.3. Section 5.4 describes the implementation techniques and tools employed for developing the two versions of the system. Section 5.5 describes some application-independent and application-related measurements to assess the performance of the two implementations of the model. Section 5.6 summarizes our experience with the two implementations so far.

5.2. Developing Applications Using the FrameWorks System

From a user's point of view, the process of developing a distributed application is identical with respect to the *FrameWorks(R)* or the *FrameWorks(G)* systems. Also, the high-level architecture of the two versions is quite similar. However, the two implementations differ significantly with respect to low-level details. In this section, an overview of the process of developing a distributed application using the *FrameWorks* system is presented. The underlying details of this process, as well as various components of the system, are described in the next section. The discussion in the present as well as the following section is applicable to both versions.

For generating a distributed application, two types of inputs are required from a user. First, the user prepares source code modules as described in Chapter 3. These modules, called *FW* modules, are written in an extended version of the C language that allows *call* and *reply* statements. The source code for modules is stored in files that have ".f" suffixes. The second type of input consists of the preparation of the application graph for the application which specifies assignment of templates to each module, the specification of their interconnection, and the names of the input and reply frame(s) for each node of the graph. The interactive graphic interface *Module_Craft* is used for this purpose. Using the command and item menus of *Module_Craft*, a user prepares and edits application graphs. Allocation constraints of a module can also be specified using *Module_Craft* via its *Allocation* command. The *include* and *exclude* options allow the specification of processors that may be selected or avoided for allocating a given process. The details of the *Allocation* command are presented in the following section.

After preparing the source code modules and the application graph, the command

```
fwconfigure application_name
```

may be used. This command causes preparation of a distributed design skeleton for the application and allocation of processors to processes. Following this, the command

fwpp application_name

invokes the *FrameWorks* pre-processor that prepares C language modules from the *FW* modules prepared by the user. Next, the command

fwcompile application name

compiles the C modules and links them with the run-time library of *FrameWorks*. Finally, the application is executed by simply invoking the main module of the application.

Of course, it is possible to put the above three commands in a single executable script. However, keeping them separate does serve a useful purpose. For example, if only processor allocations are changed then only the *fwconfigure* command needs to be re-executed. Similarly, if there are changes to the source code of the modules, then only the last two commands need to be re-executed.

5.3. Components of the FrameWorks System

This section presents a detailed description of various components of the *FrameWorks* system. Figure 5.1 shows various components of the system that are used for developing a distributed application. A *FrameWorks* application is executed in the presence of an *Execution Time Monitor (ETM)*. The complete environment in which an application is executed is shown in Figure 5.2. The shaded boxes represent various programs that belong to the *FrameWorks* system. The components shown in Figure 5.1 and Figure 5.2 may, in some cases, have different functionality for the two versions of the system. Whenever this is the case, the differences between the two versions will be described.

5.3.1. Module_Craft

As mentioned earlier, *Module_Craft* implements a visual sub-system for initial development and editing of application graphs. A sample *Module_Craft* screen layout is shown in Figure 5.3. Icons for various templates are shown as a horizontal menu

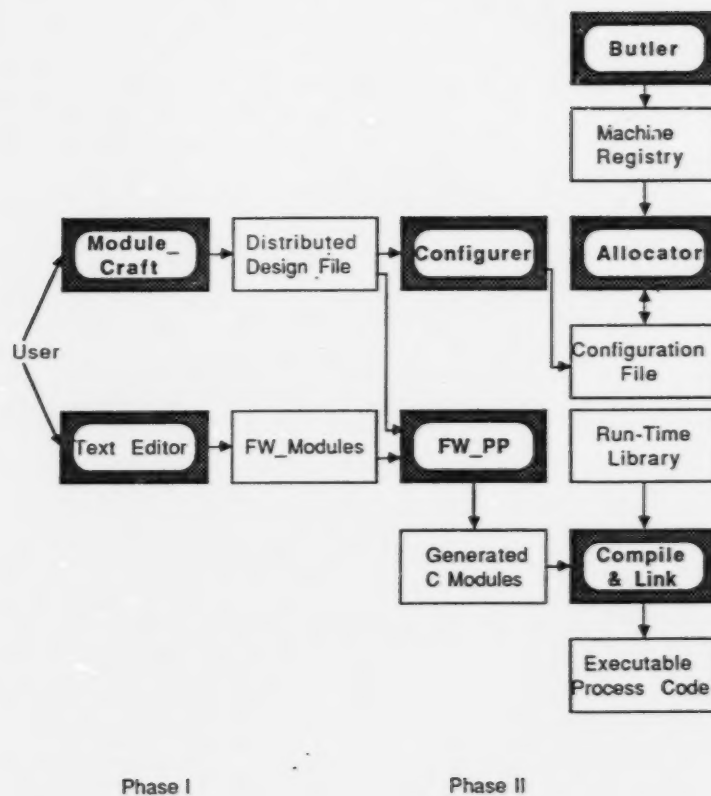


Figure 5.1. Developing Distributed Applications Using FrameWorks

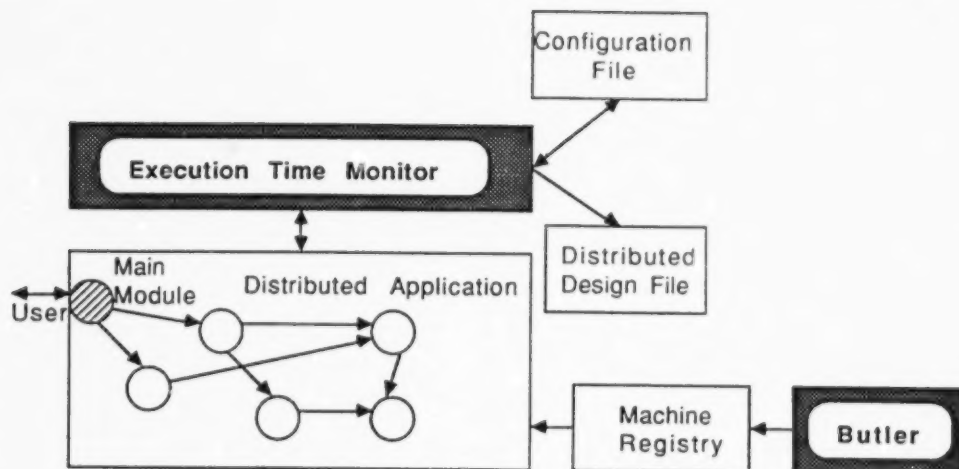


Figure 5.2. Execution Time Environment

near the bottom end of Figure 5.3. The vertical menu at the right hand side shows the available commands.

The *Load* command allows an already developed applications graph to be displayed and, if desired, edited. The *Add Input Template*, *Remove Input Template*, *Add Output Template*, *Remove Output Template*, *Add Body Template*, and *Remove Body Template* commands are used for adding or removing a specific type of template from a given module. The *Add Input Template* command requires the programmer to specify the name of the module associated with the node. A link between two nodes can be added or removed using the *Add Link* and *Remove Link* commands respectively. The *Add Link* command requires specification of the names of *input* and *reply* frames associated with the given link. The *Allocation* command is used for specifying the constraints on allocating a processor for a selected node. This command is described below in detail. The *Query Link* command is used for displaying the names of the frames associated with the specified link. Similarly, the *Query Process* command displays the name of the module associated with the selected node. The *Save* command can be used for saving the current state of the application graph. A currently active application graph can be removed from the display and memory of *Module_Craft* using the *Close* command. This command facilitates work on more than one application graph during a single session with *Module_Craft*. Finally, the *Quit* command is used for ending the current session of work with *Module_Craft*.

Allocation constraints of a module can be specified using the *Allocation* command. The *include* and *exclude* options allow the specification of processors that may be selected or avoided for allocating a given node. These commands may be repeated. The special name *free_pool* denotes all the processors on the network other than those specifically mentioned using the *include* and *exclude* options. The list of preferences are scanned strictly sequentially from top to bottom and, once a selection is made, the rest of the list is left unscanned. Using these options, a user may specify his processor allocation constraints completely, partially, or may leave the processor selection

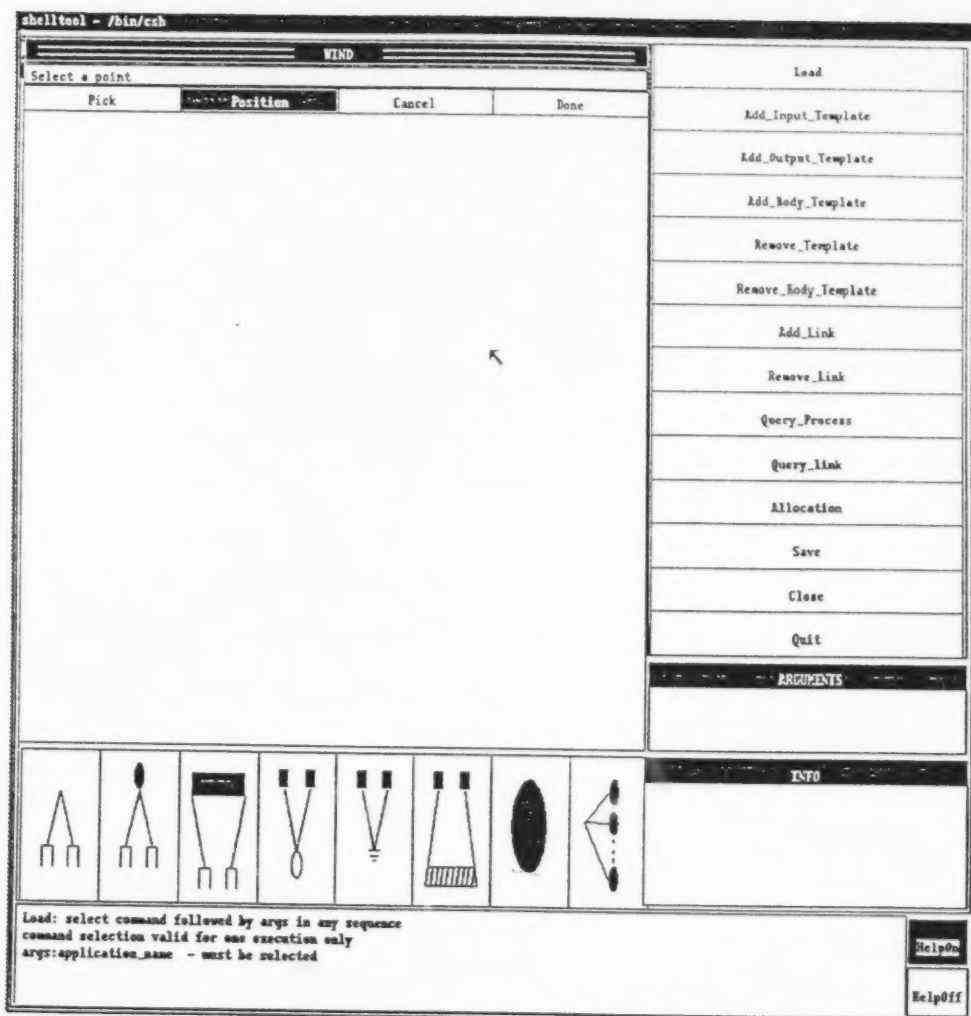


Figure 5.3: Screen Layout of Module_Craft

entirely up to the *FrameWorks* system. For example, let us assume the following sequence of preferences are specified for a non-replicated node *X*:

```
include sun001;
include sun005;
exclude sun013;
include free_pool;
```

The allocator, in this case, first attempts to allocate *sun001* or *sun005*. If neither of them is available, any processor except *sun013* may then be selected from the environment.

The above set of allocation constraints is an example of a partial specification of processor allocations. The user may leave the allocation entirely to the system by specifying no allocation constraints. The system then selects processors from the environment based on their load status. Processor allocation can be specified completely for the above example through the following set of constraints:

```
include sun001;
exclude free_pool;
```

In this case, no other processor except *sun001* will be allocated for the process. Module_Craft provides another option, *replicas*, for specifying the extent of replication for a node, in case the node meets the conditions for replication. The syntax for this option is:

```
replicas Number
```

In case of a manager template, a fixed number of processes, as specified by *Number* are executed. In case the replicated node has a *contractor* type body template, *Number* is taken as the maximum number of replications of the node that can be present at the execution time. In this case, the actual number of replications present at any time during execution depends on the amount of in-coming work to this node as well as the number of available processors.

This processor allocation scheme is simple yet quite powerful. It provides a significant level of flexibility in the selection of processors that are used during a particular execution. It can also be used to specify the number of employee processes for a replicated node that can be created at execution.

5.3.2. Text Editor

A text editor can be used for preparing the source code for various modules in the application. A user may use any text editor of his choice for this purpose.

5.3.3. Configurer and Allocator

Currently, *Configurer* and *Allocator* have been implemented as a single component which is invoked using the

fwconfigure application-name

command. These two components are responsible for preparing a detailed configuration of the distributed application and allocation of processors to processes. The information in the application graph is stored in a file called the *Distributed Design File*. This information is used by the *Configurer* and *Allocator* to prepare a new file called the *Configuration File*. In the configuration file, each process of the application is assigned a logical identification number. The *Allocator* augments the information in the configuration file by assigning processors to processes. Processors are allocated in a depth-first manner starting from the *main* module. For a module that has not been assigned the contractor body template, the selection of processors can be made just prior to initiating execution of the application. By invoking the *fwconfigure* command prior to an execution, depending upon availability of processors, during different executions different processors may be selected. If the allocator cannot satisfy a user's allocation requirements with the current status of processors, the allocation procedure is aborted and the user is given an appropriate error message. The user then has two options. He can modify his allocation constraints so that they can be satisfied with the currently available processors, or he can retry the allocation

procedure later when the processors required by him are available.

For a contractor module, the processors are selected for employee processes at execution time. These processors are also selected on the basis of allocation constraints specified by the user for the given node. In this case, the number specified via the *replicas* option of *Allocation* command is taken as the maximum number of employees that can exist at any time during execution.

5.3.4. FW_PP

The information in the *Distributed Design File*, along with the application's source modules, is processed by the *FrameWorks* preprocessor (*FW_PP*). *FW_PP* inserts code within the source code of each *FW* module prepared by the user to achieve proper synchronization and communication. To do this, the user executes the

fwpp application_name

command. Primarily, code is inserted to achieve the following functions:

- 1) Establishing connections with neighboring modules and the *Execution Time Monitor*,
- 2) Communication of user and control messages necessary for executing *call* and *reply* statements,
- 3) Carrying out the client-server execution loop as described in the previous chapter, and
- 4) Carrying out the termination procedure for the node.

The exact contents of the inserted code depend upon the templates that are assigned to a node and, to some extent, on the templates of the neighboring nodes. For example, protocols for communicating with a node that has a *contractor* body template are different from protocols for communicating with a node that has an *empty* body template. The output of *FW_PP* consists of C language modules.

5.3.5. Compilation and Linking

Most of the code that is required to execute a *FrameWorks* application resides within the run-time library of *FrameWorks*. The code inserted by the *FW_PP* consists of calls to routines in the run-time library. As mentioned earlier, when the user executes the

fwcompile application_name

command, executable modules are prepared by compiling these modules and linking them with the run-time library of *FrameWorks*.

5.3.6. Execution Time Monitor

As shown in Figure 5.2, an application runs in the presence of an *Execution Time Monitor (ETM)* and *Butler* (See Section 5.3.7). The *Butler* maintains a database that contains information regarding the load-status of various machines in the environment. It is described in detail in the following subsection. The purpose of *ETM* is to oversee the execution of a *FrameWorks* application.

In the restricted model, the *ETM* carries out the initial hand-shake procedure to ascertain that all processes in the application have started properly. In the general model, however, the *ETM* performs many more functions:

- 1) It carries out the initial hand-shake procedure to ascertain that all the processes in the application have started properly.
- 2) It maintains detailed status information about all the processes in the application and coordinates the execution of *call* and *reply* statements.
- 3) It coordinates the termination procedure for the entire application.
- 4) Optionally, it can collect an ordered history of events during execution for the application. A system component, *View_Graph*, uses this information for the purpose of post-execution analysis (See Section 5.3.8).

5.3.7. Butler

The nodes in an application that have a *contractor* type body template require information about the load-status of processors on the network in order to take a decision regarding acquisition of processors. The system component *Allocator* also requires the same information for binding processors to various processes in the application. The system component *Butler* is used to aid in this process. *Butler*, on demand, gathers information regarding the load-status of processors on the network. A system file called *m.db* contains the names of all the processors that can possibly be accessed by any *FrameWorks* application. The *Butler* gathers information about each processor in the *m.db* database which indicates whether the machine is currently *up* or *down*, and three integer numbers that indicate the average length of the run queue during the last 1, 5, and 15 minutes. This information about processors is stored in a file called *machine_register*. Before execution, the *Allocator* uses the information in the *machine_register* for allocating processors to processes. During execution, each contractor process uses this information along with the allocation constraints specified by the user for the purposes of hiring and firing employee processors.

5.3.8. View_Graph

This section presents a post-execution analysis tool, called *View_Graph*, which can be used for algorithm animation, performance monitoring, as well as detecting deadlocks in *FrameWorks* applications. A preliminary implementation of *View_Graph* exists. Figure 5.4 shows the visual interface of *View_Graph*. This section presents the complete design of *View_Graph*. The discussion is aimed to illustrate the kind of additional services that can be provided via the *Execution Time Monitor* of the general model.

The *Execution Time Monitor* in the *FrameWorks(G)* system can capture significant events and other performance data during execution by logging them locally as well as within each of the processes in the application. Basically, the types of data

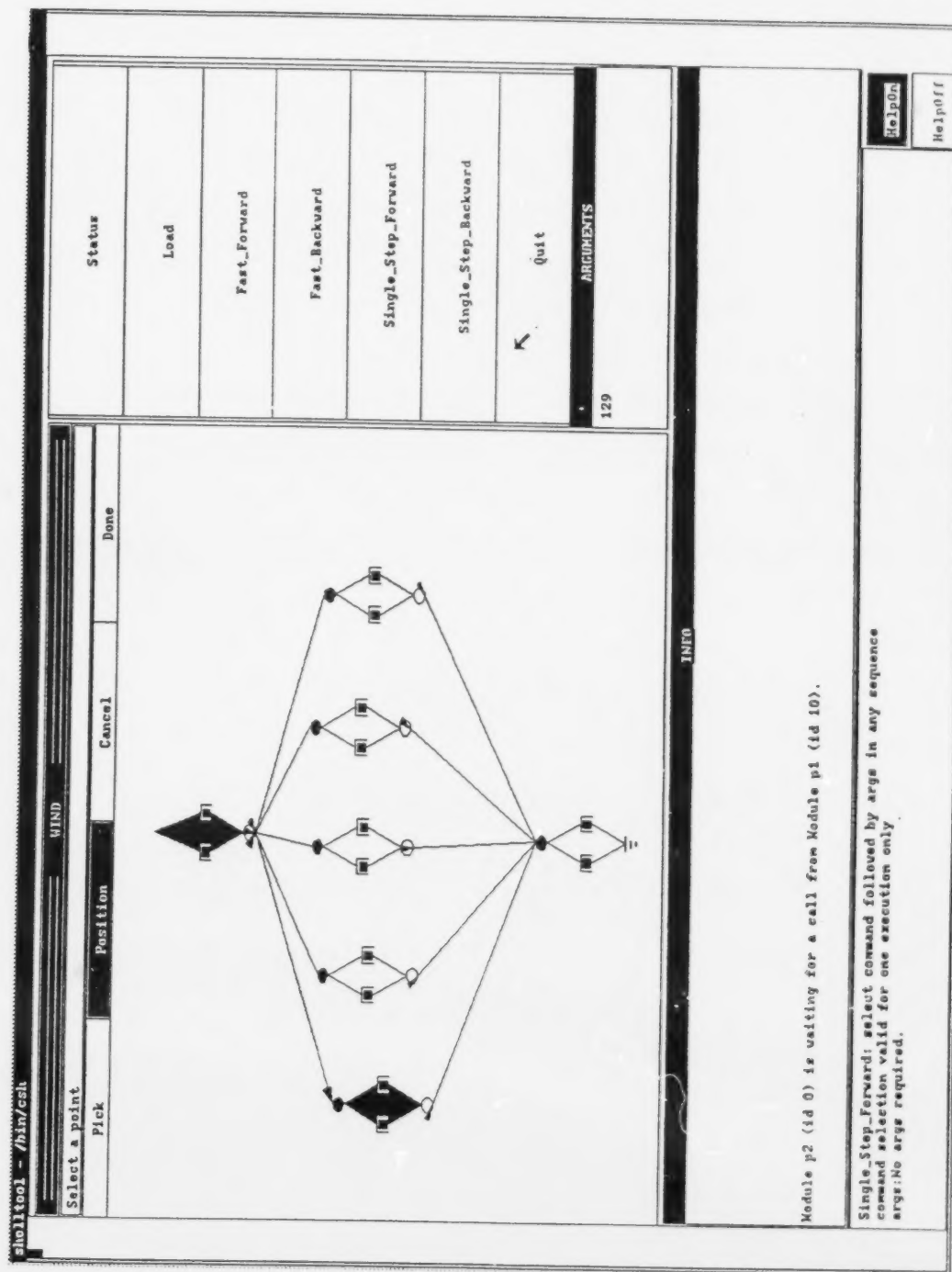


Figure 5.4: Screen Layout of View_Graph

collected can be put into three categories:

- 1) Logging of such events as arrival of a call and request for call. Such data is collected in part by each process locally and in part by the *ETM* at the global level.
- 2) Performance statistics, for example, the number of calls received, total time waiting for a call, or time spent waiting to call some other process. Such data is collected only within a process.
- 3) Output trace information that is specified to be collected by the user. The request is made in the form of output statements within the source code of each module. The trace information is recorded in a file by each process, and can be selectively displayed, if desired, using *View_Graph*.

After execution, *View_Graph* post-processes the collected data and the complete sequence of events can be displayed in a deterministic, controlled, and interactive manner. Modes of display can be adjusted to slow or fast, backwards or forwards. On workstations with color monitors, the color of a process icon or edge can be used to display the changing status of a process. The *Status* command allows the display of performance statistics for a particular process or the display of trace information between selected events. Although the development of *View_Graph* is presently in a preliminary stage, it is considered significant for the following reasons:

- 1) It is a component which is tightly integrated with our computational model rather than a facility added as an after thought. The same application graph, prepared by the user, is used for development as well as post-analysis.
- 2) As in [4], the user does not have to create any application-specific visual interface for the purpose of post-execution analysis. Other than specifying the trace information to be collected at the execution time, the user simply takes no other part in algorithm animation and performance monitoring.
- 3) Since it is a post-analysis system, the display process itself does not interfere with, or become a bottleneck, in the execution of an application.

5.4. Implementation Tools and Techniques

FrameWorks has been implemented mostly using tools developed in our department. For the distributed computing aspects, the Network Multiprocessor Package (NMP) is used [76]. NMP is built using sockets [71] and it provides a library of high-level routines to support remote process initiation and message passing among processes. The graphic interfaces for the system have been developed for color as well as monochrome monitors using locally developed user interface management tools called, Diction, Chisel, and Vu [101, 102]. Visual interfaces for *Module_Craft* as well as *View_Graph* have been implemented using these tools.

The implementations of the *FrameWorks(R)* and *FrameWorks(G)* systems are based on the restricted and general implementation models respectively. These models were described in the previous chapter. The dynamic creation and termination of employees in the case of a *contractor* template is achieved by implementing the template as a closely related group of processes consisting of a coordinator and its employees. The implementation is quite different from the implementation of threads in Mach [88] in the sense that the coordinator and its employees do not share an address space. It is more similar to the "process-groups" of V [23] with the significant difference that the management of employee processes is transparent to the programmer, and is handled completely by the coordinator process. For every module which has been assigned a contractor body template, a separate coordinator process is started. This coordinator process is responsible for managing the employee processes. From the users' point of view, an employee process represents an instance of the user module functioning as a server. The actual number of employee processes at any time during execution depends upon the allocation constraints specified by the user, the available number of processors on the network, and processing requests that arrive from client nodes. In addition to managing the number of employee processes, the coordinator is responsible for receiving input from the input nodes, sending it to a selected employee, receiving the input back from the employee, and sending the output to output nodes.

An employee repeatedly serves calls assigned to it by its coordinator process until it is ordered to terminate. Thus, in addition to providing the complete functionality of a contractor template, our implementation scheme avoids the inefficiency of starting a new process for each call and then terminating it after the call is served.

5.5. Preliminary Experimental Results

In this section, we present some application-independent as well as application-related measurements taken with the two versions of the *FrameWorks* system. These experiments were performed on diskless Sun-3 workstations, sharing a Sun 3/180 file server and a Sun 3/60 page server. The workstations run under the Sun OS 3.5 and communicate over 10 Mbps ethernet interfaces. Some performance results of the distributed radiosity system described in Chapter 3 are also described. These results are not meant to be comprehensive. Rather, they are presented to give an insight into the performance characteristics and overheads of *FrameWorks*, as well as some indication of speed-ups that may be attained by using the system. These results also yield some interesting insights into the relative performance characteristics of the two implementation models described in the previous chapter.

5.5.1. Application-Independent Results

The first set of experiments were done to ascertain the cost of executing *FrameWorks*' blocking and non-blocking *call* statements with varying message (frame) sizes. These results are based on a variety of configurations, as shown in Figure 5.5. Although, for the sake of simplicity and brevity, not all possible combination of templates are used, all different types of templates are represented in these configurations. The time taken for 100 calls was measured for each configuration. Each experiment was repeated five times and the average of the observed times was calculated in each case. The results are shown in Figures 5.6, 5.7 and 5.8. Figure 5.6a and 5.6b show the average time taken in executing 100 non-blocking and blocking calls respectively in the case of the *FrameWorks(G)* system as the message size is varied. The message

size is given by the size of input or reply frames. Similar results for the *FrameWorks(R)* system are shown in Figures 5.7a and 5.7b. Since assimilator template is called only in a non-blocking mode, in Figures 5.6b and 5.7b, there are no curves showing results for blocking calls for configuration (ii).

There are several observations that can be made regarding these results. First, the cost of executing a *call* statement for templates that are not concerned with the run-time management of replicated instances of a user's module is small (approximately 0.02 seconds). However, the contractor template, which manages replications dynamically at run-time, is more expensive. In each case, the overhead is because of two factors: the execution of extra code associated with the template, and the exchange of control messages for management purposes. It should be noted that although templates add overhead in terms of extra code and messages, a similar overhead will be there even if the user himself writes code to achieve similar functionality as provided by templates. Except in the case of expert users, this overhead is not likely to be smaller than overheads associated with *FrameWorks'* templates. Therefore, although the overhead would exist whether the code is hand written or generated using templates, *FrameWorks* simply provides an easy way to get quick and correct code.

Second, the cost of executing a blocking call is generally higher than executing a non-blocking call. However, this difference in costs grows larger as the size of input and reply frames is increased. For large frame sizes, the cost of data transfer dominates the overall cost, whereas for small frame sizes the number of exchanged messages dominate the overall cost.

Third, the communication in the restricted model is somewhat more efficient than the communication in the general model. The reason for this is the higher number of control messages exchanged in the general model. This is illustrated in Figure 5.8 which compares the cost of 100 non-blocking calls using configuration (i) for both models. The general model exchanges about four to five times more control messages than the restricted model. However, an interesting aspect of the results is that the

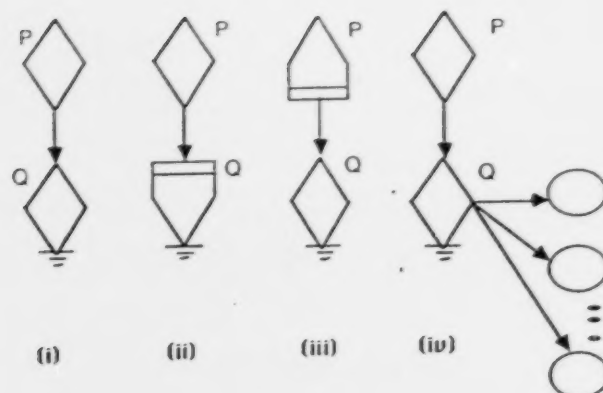


Figure 5.5: Experimental Configurations for Figure 5.6-5.8

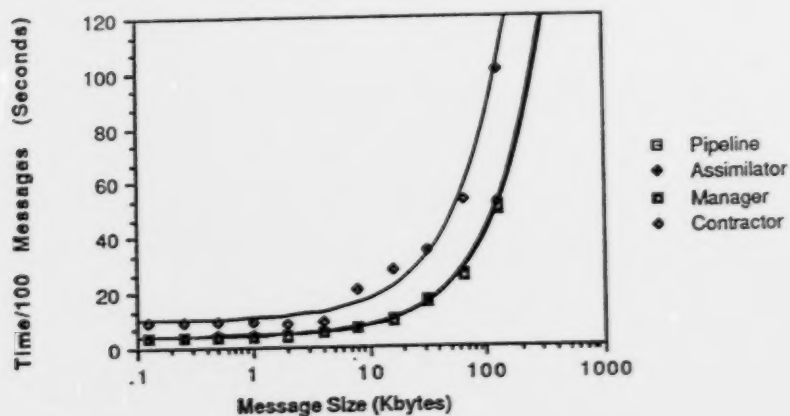


Figure 5.6a: Cost of Non-blocking Calls in FrameWorks(G) System

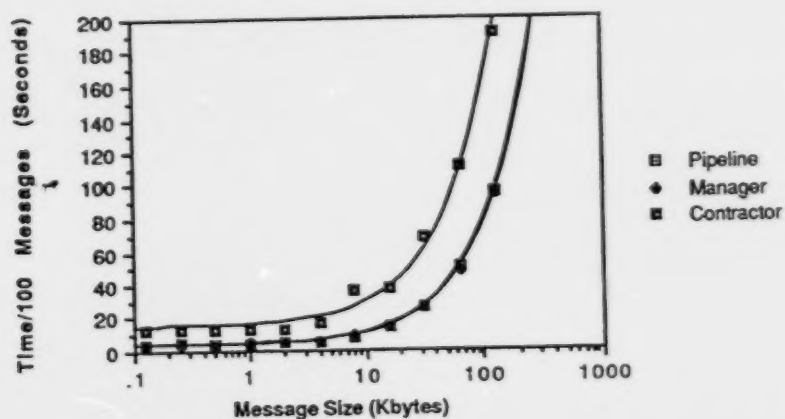


Figure 5.6b: Cost of Blocking Calls in FrameWorks(G) System

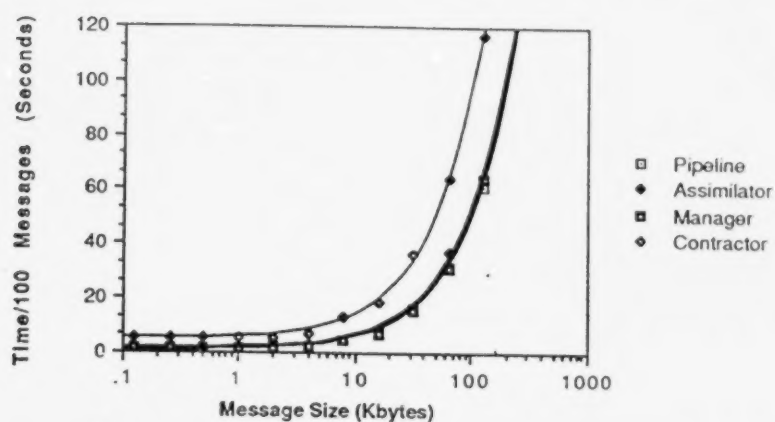


Figure 5.7a: Cost of Non-blocking Calls in FrameWorks(R) System

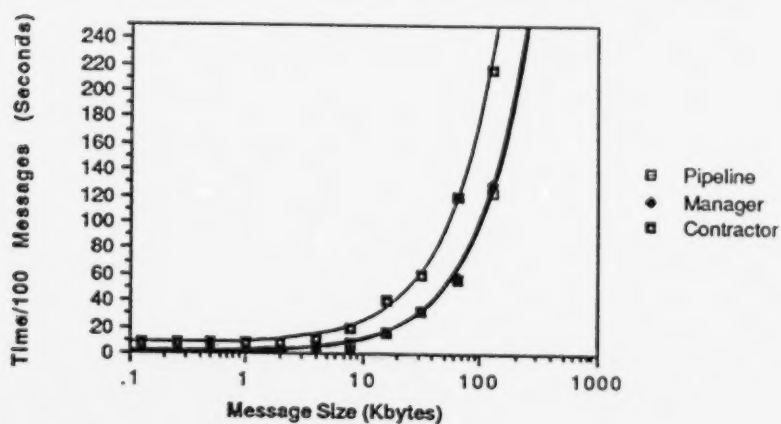


Figure 5.7b: Cost of Blocking Calls in FrameWorks(R) System

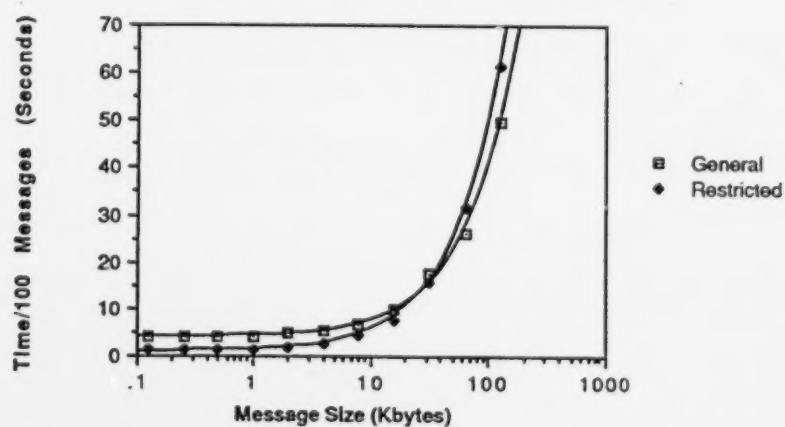


Figure 5.8: Comparison of Costs for the Two Implementation Models

increased overhead in the general model is much less than what would be expected merely on the basis of counts of control messages. The possible explanation for this could be the distributed activity of the exchange of control messages, or the internal characteristics of the TCP protocols.

The second set of experiments were done to determine the behavior of templates that manage replication, namely the contractor and the manager. Configurations used for these experiments are shown in Figure 5.9. In each case, the module P iteratively makes non-blocking calls to module Q. The module Q repeatedly computes the floating point multiplications of two variables. The time taken in executing the module Q is controlled by the number of times this multiplication is done. This time represents the granularity of the work done by the replicated processes. In our experiments, the number of multiplications done in module Q is adjusted to obtain a granularity of approximately 0.5 second or 1 second. In the case of experiments with 1 second grain size, the module P makes 2000 calls to module Q. In the case of experiments with 0.5 second grain size, the number of calls made by P to Q is doubled to 4000. This allows us to keep the time taken for sequential execution to be the same for each experiment.

The results of these experiments are shown in Figure 5.10, 5.11, and 5.12. For the sake of comparison, the average time taken when the same task is done sequentially is also shown. Figure 5.10a shows the variation of real time taken in completing the application as the number of employed workstations is varied from one to ten. These results are based on the *FrameWorks(G)* system and configuration (i) of Figure 5.9. Results are shown for 1 second grain size as well as 0.5 second grain size. The results when the contractor template is used (configuration (ii) of Figure 5.9) are shown in Figure 5.10b. The comparative performance of manager and contractor templates for the case of 0.5 second grain size is displayed in Figure 5.10c. Similar results for the *FrameWorks(R)* system are shown in Figure 5.11a, 5.11b, and 5.11c respectively.

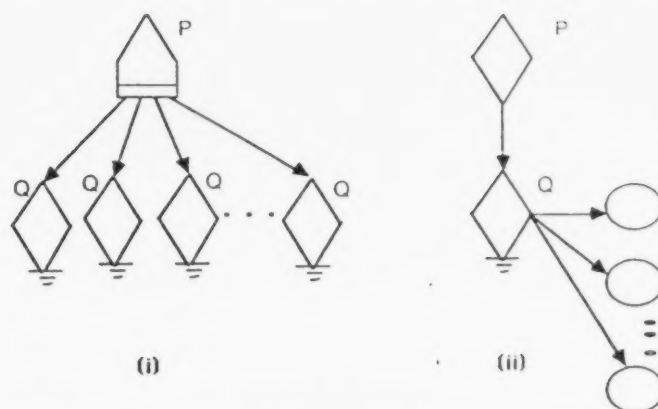


Figure 5.9: Experimental Configurations for Figures 5.10-5.12

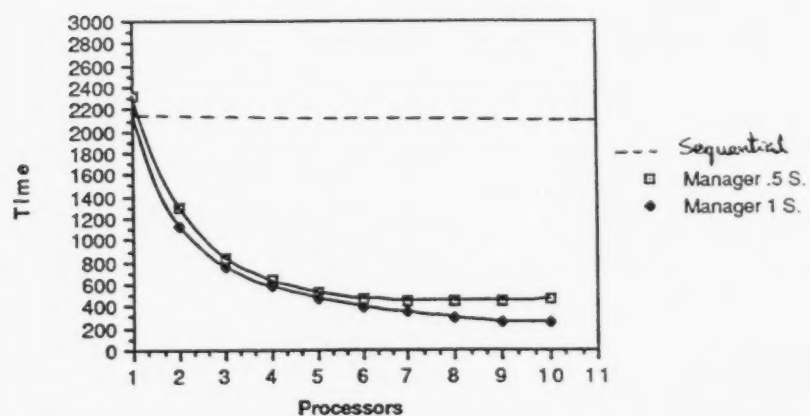


Figure 5.10a: Performance of Manager Template in FrameWorks(G) System

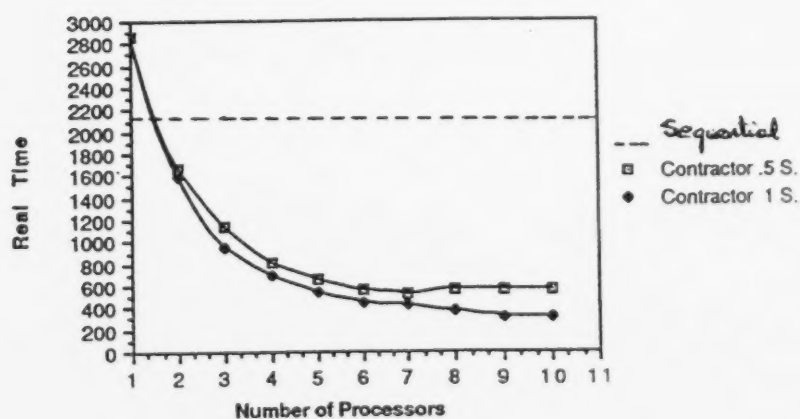


Figure 5.10b: Performance of Contractor Template in FrameWorks(G) System

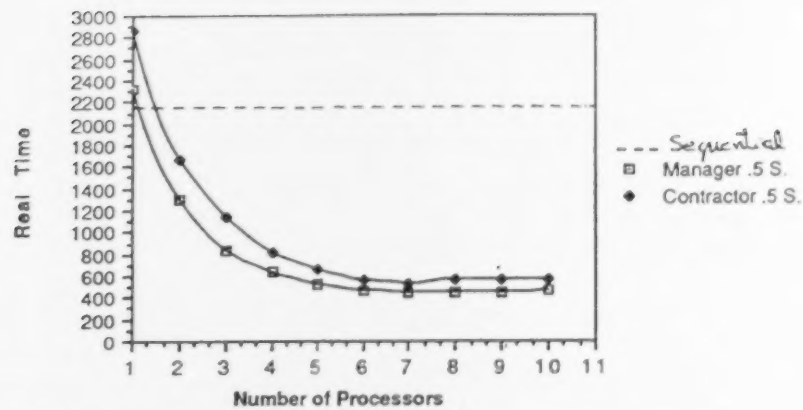


Figure 5.10c: Comparison of Manager and Contractor Templates for FrameWorks(G)

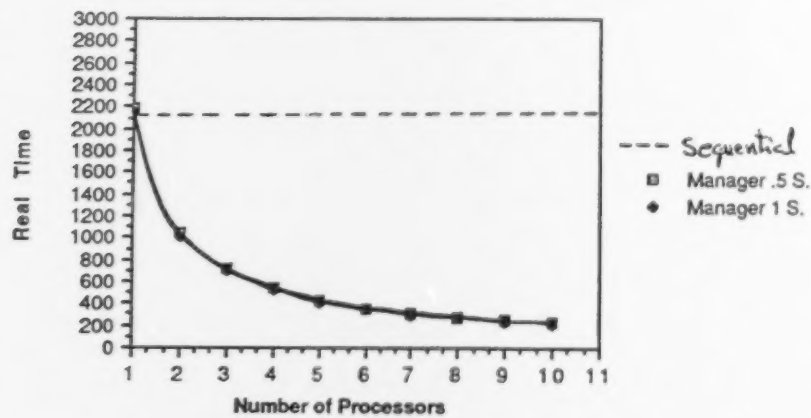


Figure 5.11a: Performance of Manager Template in FrameWorks(R) System

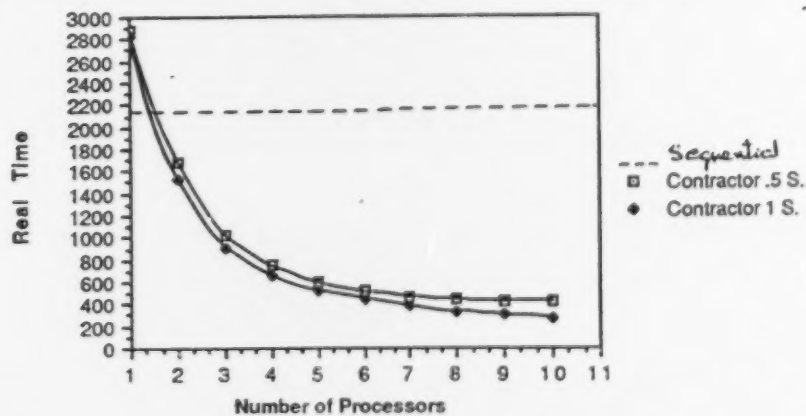


Figure 5.11b: Performance of Contractor Template in FrameWorks(R) System

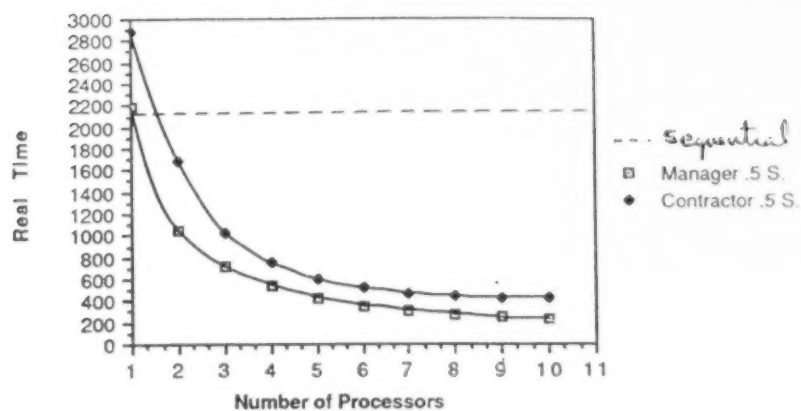


Figure 5.11c: Comparison of Manager and Contractor Templates for FrameWorks(R)

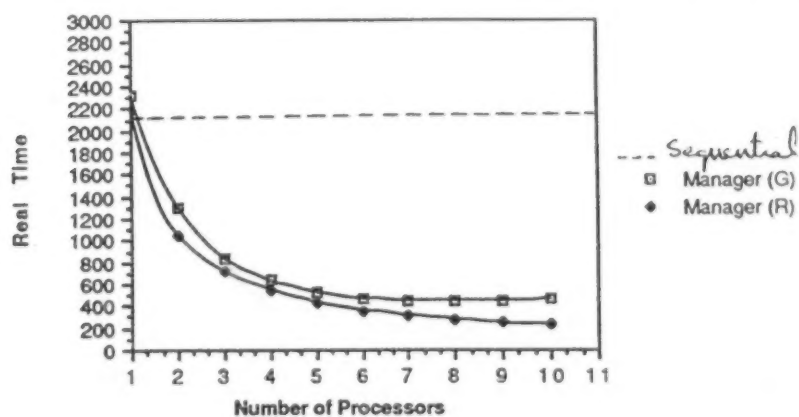


Figure 5.12a: Comparison of Manager Template's Performance

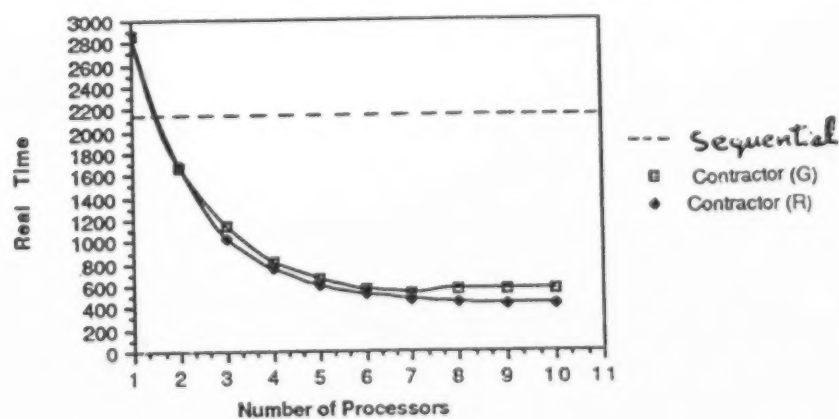


Figure 5.12b: Comparison of Contractor Template's Performance

These results suggest that the overheads of a contractor are always higher than the overheads of a manager. Both contractors and managers perform well in cases where the amount of work done on each call is one second. When the work done on each call is reduced to 0.5 second, the contractor does not do very well, but the manager still scales reasonably well. Although, the contractor's overhead are higher than those of manager's, in our opinion, the advantages provided by the contractor template in an environment with changing processor resources justify its inclusion in our selection of templates.

The comparative performance of managers and contractors in the *FrameWorks(R)* and *FrameWorks(G)* systems is shown in Figure 5.12a and 5.12b respectively. These results show that the manager and contractor in the restricted model are more efficient than the manager and the contractor respectively in the general model. However, once again the difference in their relative efficiency is not as much as would be expected merely on the basis of the number of exchanged control messages in each case.

The results in Figures 5.10-5.12 show the overhead costs and range of acceptable grain sizes for good performance. Coarse grain applications, where computation times of modules are of the order of several seconds (refer to the application in the following subsection), and often even few minutes are not rare. In such applications, the grain sizes are well within the good performance range for the contractor as well as the manager.

5.5.2. The Distributed Radiosity System

In this subsection, some performance data taken using the distributed radiosity system described in chapter 3 is presented. In addition to the sequential version, all three distributed configurations discussed in chapter 3 were implemented. These configurations are shown together in Figure 5.13. Figure 5.14 shows real time taken in computing *formfactor* values for 120 patches of a particular image for each of the configurations. The variation in real time in the case of configuration C as the number

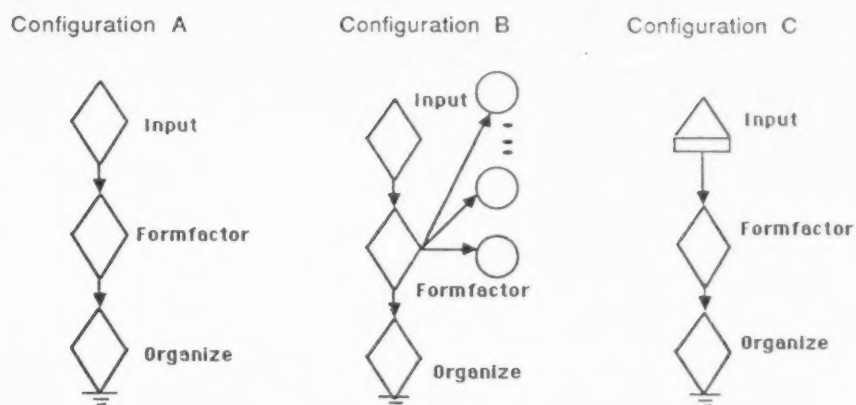


Figure 5.13. Three Configurations of the Radiosity System

Configuration	No of Processors	Execution Time (In Seconds)	Speed-Up Ratio
Sequential Version	One	5400	1
Configuration A	One	5440	1.0
Configuration A	Three	5017	1.1
Configuration B	Five (Maximum)	1355	4.0
Configuration B	Ten (Maximum)	661	8.2
Configuration C	Ten	541	10.0

Figure 5.14. Speed-ups Achieved with Various Configurations

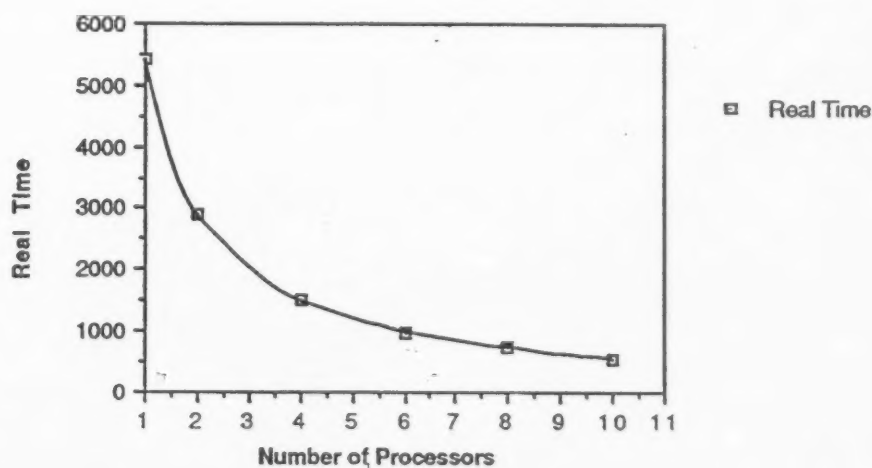


Figure 5.15. Variation of Real Time with Number of Processors for Configuration C

of processors is varied from one to ten is shown in Figure 5.15.

The results in Figure 5.14 show that the degradation in the performance of the application is small even when the pipelined parallelized version (as shown in configuration A of Figure 5.13) of the application is executed using a single processor. However, only a small speed up is observed when each of the module is executed on a separate processor. This is because most of the work done in the application is done by the *formfactor* module. Therefore, other modules of the pipeline have to keep waiting for it to either receive input or send output. When the *formfactor* module is replicated using a contractor (configuration B of Figure 5.13) or a manager (configuration C of Figure 5.13) template, significant speed-ups are observed as the number of processors employed is increased.

Although this example is a simple one and is expected to get good performance, these experiments demonstrate that it is possible to achieve a fair amount of speed-up using *FrameWorks*. The absolute value of the speedup is not important; given that our graphics group ran this program sequentially, *any* improvement in performance was welcome. That they achieved this with minimal programming effort was equally welcome. Distributed configurations, such as *B*, *C*, and *D*, have fairly complicated distributed structures operating behind them. Writing such an application using low-level primitives, such as sockets or RPCs [9,71], would require considerable effort on the programmer's part. Since such an effort would be replicated, perhaps unwittingly, by other programmers while writing different applications, *FrameWorks* provides an easy way of avoiding this wasted effort.

5.6. Conclusions

The chapter presented the implementation of the *FrameWorks* system. The practical feasibility of the both of the implementation models was demonstrated. Our preliminary experimental results indicate that the restricted model is slightly more efficient than the general model. This difference is important for applications that exploit finer

grain parallelism (grain size less than 0.5 second approximately). However, for large grain applications, the difference may not be significant. The information that can be collected within the *Execution Time Monitor* of general model can be used for providing a variety of diagnostic services to the user. These considerations, coupled with the fact that there are other restrictions in the restricted model, make the general model more attractive for future considerations.

Chapter 6

Enhancements to FrameWorks Approach

6.1. Introduction

Previous chapters describe a new approach to developing parallel applications in a workstation-based environment. In summary, the work covered so far consists of the description of the *FrameWorks* model, formal specification of its templates and implementation models, and the description of the two versions of the *FrameWorks* system. During the course of the development of the model as well as the system, several design decisions and choices had to be made. Three major factors that influenced our decisions in order of their importance were:

- 1) our original goal of providing a model for parallel computing that suits the needs of a workstation environment,
- 2) the hardware architecture and the software environment that were available to us for development and experimentation, and
- 3) constraints on other resources such as time and financial costs.

The present chapter reviews the rationale behind the important decisions and their impacts on the final design. Also, several extensions and variations of *FrameWorks* are described. The issues discussed in this chapter fall under two separate categories: those related to the model itself, and those related to the implementation of the model. Sections 6.2 and 6.3 discuss various issues that fall into each of these categories. Section 6.4 describes our development environment and discusses some customized tools that were used for the development of the system. Section 6.5 outlines our experience with the *FrameWorks* system so far. Concluding remarks are presented in section 6.6.

6.2. Variations of the FrameWorks Model

The separation of application code from the specification of parallel structuring, as provided by the templates, is one of the basic aspects of the *FrameWorks* model.

In this thesis, the concept of template-attachment was used to develop a system that would facilitate the generation of parallel applications in a workstation environment in a easy and flexible manner. However, it appears that the template-based approach is more general and it can be applied for developing other tools for the development of parallel applications. In this section, some potential variants of the *FrameWorks* model are described. All variations are based on the template-based approach. Section 6.2.1 describes hierarchical structuring of applications using the templates. Sections 6.2.2 and 6.2.3 discuss variations that are suitable for specific classes of applications. Section 6.2.4 discusses the applicability of the model for the general MIMD environment.

6.2.1. Hierarchical Node Organization

In this section, the possibility of representing multilevel application graphs through ordinary *FrameWorks* application graphs is considered. It seems that a node in a computational graph may further be decomposed into complex multilevel subgraphs with many nodes, thus allowing an application to be developed in a hierarchical manner [97]. A node can either be assigned a module, or it can be a reference to a subgraph in a library. This design method is consistent with the already established practices of top-down structuring and stepwise refinement. It also facilitates development of component libraries whose components can be used whenever required. However, such representation requires some design rules to be followed:

- 1) A *FrameWorks* node can be replaced by a subgraph that has a single node for receiving input and a single node for sending output. In the substituted subgraph, the input template of the node receiving input must be the same as the input template of the node being replaced. Similarly, the output template of the node sending output must be the same as output template of the node being replaced.
- 2) Body templates are assigned to nodes at the lowest level of the expanded graph.

As an example, consider the distributed-radiosity application of Chapter 3. The

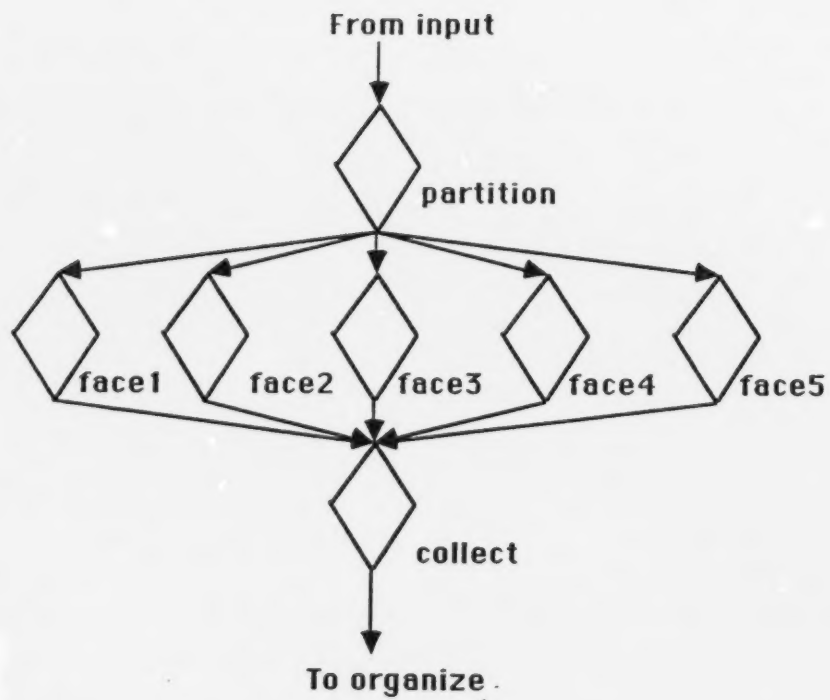


Figure 6.1: Hierarchical Node Organization

formfactor node in this application can be broken down into the subgraph shown in Figure 6.1. Here, the node *partition* divides the computation for a given patch into subproblems for each of the five faces of the hemi-cube around the given patch. Nodes *face1* to *face5* perform this function. Finally, these computations are combined together by the node *collect*. Now, to speed up the computation process the nodes *face1* to *face5* can be assigned *contractor* body templates.

6.2.2. Automatic Sequential/Parallel Conversion

One of the goals of *FrameWorks* is to allow restructuring of an application with minimal effort. An application may require restructuring to execute it with a different number of processors, or to tune it for higher performance, or even to convert it into the sequential form. This section discusses certain types of applications for which it might be possible to automatize the process of conversion between the sequential and parallel versions. Conversion to the sequential form allows a developer to conduct the initial development and debugging in the more familiar sequential environment. When the sequential version of the application is complete, its parallel version can be automatically generated.

As pointed out earlier, the application for distributed radiosity, as discussed in Chapter 3, can be easily modified to generate its different configurations. The example can also be converted to the sequential form by merely modifying the remote procedure call statements in *formfactor* and *organize* modules. The required modifications can also be handled automatically using a tool that removes the *call* keyword from the remote procedure call statements. In fact, in our implementation, the conversion is handled through the use of a C-preprocessor variable, that allows the modules to be compiled slightly differently depending upon whether the sequential or parallel version is required.

However, there are applications for which the problem of conversion between sequential and parallel versions is more complicated and requires further investigation.

For example, Figure 6.2a shows an application consisting of three modules. The application graph has a cycle. Let us suppose that in one particular sequence of procedure calls, the module A calls the module B, the module B calls the module C, and the module C calls the module A. Now, if all the calls are of blocking type, then the sequential version of the application involves recursion. However, in the parallel case, this would result in a deadlock situation because modules A, B, and C wait for each other in a cyclic manner. The presence of a cycle in the application graph does not necessarily cause any problem. For example, if at least one of the calls in the above example is non-blocking, no deadlock would occur. Therefore, a tool handling automatic conversion between sequential/parallel versions will have to detect such situations and appropriately warn the user. For example, Taylor's work on general static analysis of Ada programs detects potential deadlock situations in an application [110].

Another situation that might require special attention occurs when an *assimilator* template is used. As an example, in Figure 6.2b, the module D waits until at least one *frame* arrives from each of the modules A, B, and C. This situation does not have an analog in the sequential case whereby several procedures together provide input parameters to a called procedure. Therefore, automatic conversion between sequential and parallel versions becomes difficult. However, it might be possible to structure the application such that different sections of source code get compiled to create the desired version. Also, it would be desirable to look into the possibility of automatically generating the pieces of source code that are required for conversion between the sequential and parallel versions.

It appears that apart from a few special circumstances, such as mentioned above, there are large classes of applications where the conversion between sequential and parallel versions can be handled automatically. However, more work is required to identify problematic situations and possible ways to handle such situations.

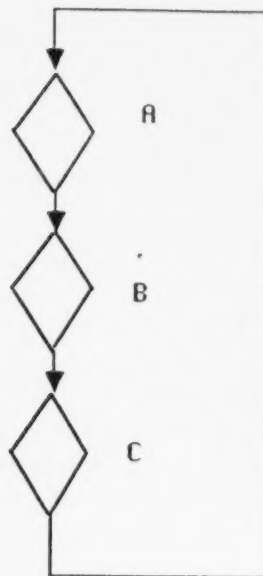


Figure 6.2a: A Cyclic Application Graph

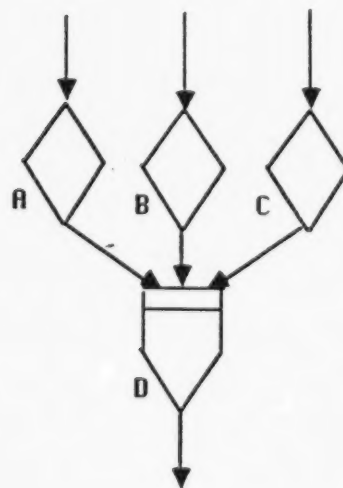


Figure 6.2b: An Application Graph with Assimilator Template

6.2.3. Contractors and Subcontractors

There is a large class of problems where the program structures are simple yet they require an enormous amount of computing. Often, in such programs, the strategy is to break the original problem down into several small independent and similar subproblems. Each of these subproblems can be assigned to a different process and the results received from these processes can be combined to form a solution to the original problem. Sometimes, the number of subproblems and their sizes can be determined at program design time. The contractor or manager templates as described in Chapter 3 can be easily used for such situations. Often, however, the problem cannot be divided into independent subproblems of equal size at program design time [6,43,77,91]. Rather, the creation and synchronization of subproblems proceeds recursively and dynamically, and these subproblems can be of widely uneven sizes. Although, the contractor or manager templates can be used in such cases as well, inefficiencies result because there could be some processors that are idle whereas the rest of the solution process cannot proceed until the results of some child processes become available (Figure 6.3). Instances of such problems are quite commonly found in problems that use divide and conquer, depth-first search, breadth-first search, or branch and bound strategies. This section describes a major potential extension of the original *contractor* template which would significantly simplify the task of developing parallel applications for such problems [99].

To efficiently handle the scheduling of dynamically evolving medium-grained tasks, it is necessary to provide two library procedures that can be used within a module enclosed in a contractor body template. The call

```
job_id = split_job(job_frame, mode);
```

causes creation of a new employee process. The structure of the *job_frame* is the same as the structure of the module's input frame. The *mode* of call can be either blocking or non-blocking. In case of a blocking *split_job* call, the execution in the calling process is blocked until a new processor is available. The call returns a positive integer

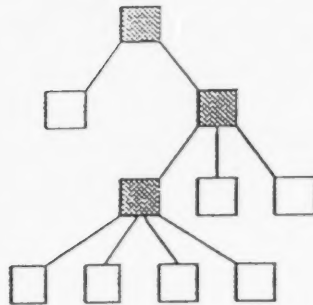


Figure 6.3: Recursive Splitting of a Task

in *job_id* that serves as the identification number for the newly started subproblem. In the non-blocking mode, the call returns immediately. If an idle processor is found, the call returns a valid job identification number. Otherwise, a negative number is returned and no new process is started. This allows an employee process to take advantage of any other currently idle employee in a dynamically expanding and shrinking pool of employees. To receive the results from processes started via *split_job* calls, another procedure *merge_job* is used. The call

result = merge_job(rep_frame, job_id, mode);

returns immediately if the non-blocking mode is used. In this case, the call returns 0 if the subproblem given by *job_id* has been completed and -1 otherwise. If the subproblem has been completed, the reply frame is returned in *rep_frame*. In the case of blocking mode, the calling process is blocked until the specified subproblem is completed, after which the call returns with the reply frame being placed in *rep_frame*.

The *split_job* and *merge_job* procedures provide a powerful and flexible method for creation and control of hierarchical threads of execution and efficient dynamic scheduling of available processors. Using these calls, the employee processes of a contractor module can further subdivide their work without worrying about the complicated processor management in the workstation environment. The *split_job* call provides a means by which a new process is not started until an idle processor is found. In such a situation, the calling process has the option of trying the *split_job* call later, or doing the job by itself. Although, the *split_job* and *merge_job* calls appear similar to other known methods of starting a new thread of execution, such as fork/join statements and methods based on spawning a new process [29,34], there are some significant differences. In these other methods, a new process is started whether or not an idle processor is available. In our case, each processor runs a single process and a new process is added only when an idle processor is available. This has several positive implications. It eliminates logically pointless and expensive context switching, and avoids clogging of systems' job queues with jobs waiting for processor time.

Also, these calls provide greater flexibility in using idle processors at execution time. The number of processes created can be adjusted according to the computational requirements of the application as well as the available processors. Allowing the idle processors to help out the more busy ones significantly reduces synchronization costs in the types of computations mentioned earlier. However, the most significant difference between these method and the scheme based on *split_job* and *merge_job* is the ease of programming applications that require dynamic creation and control of hierarchical threads of execution. As an example, the parallel chess playing program ParaPhoenix [91] performs dynamic splitting of tasks for implementing distributed alpha-beta search. It conducts a variable depth distributed game-tree search using workstations on a local area network via a carefully hand-crafted implementation. The implementation took several man-months of effort to obtain a relatively error-free program. The same functionality can be derived in a rather straightforward manner through the use of *split_job* and *merge_job* procedures.

Although, the extended *contractor* template described in this section is much more powerful than our existing *contractor* template, it was decided not to include the extended template into the present *FrameWorks* model. The primary reason for the decision is that presence of *split_job* and *merge_job* calls in the source code of a module is meaningful only when the *contractor* body template is used for the module. Therefore, the presence of these calls in the source code makes the task of restructuring an application or converting it into the sequential form more difficult. This is in conflict with the stated goals of the present *FrameWorks* model and system. However, the extended *contractor* template may save large amount of programming effort for several kinds of applications. Depending upon the overall goals, a system for parallel computation may or may not decide to implement the extended *contractor* template.

6.2.4. FrameWorks in Tightly Coupled MIMD Environments

Current design of the *FrameWorks* system is targeted towards the workstation environment. However, the *FrameWorks* model has characteristics that would make it an attractive model for parallel computing in a tightly coupled multiprocessor environment as well. In a message passing multiprocessor environment, such as the cosmic-cube [3], the model can be implemented using almost the same implementation models as used for the workstation environment. Shared memory multiprocessor environments often provide primitives for process spawning and interprocess communication which are similar to those provided by message passing environments. Under such environments, *FrameWorks* can be implemented either by using these interprocess communication primitives, or by effecting remote procedure calls and other necessary message passing activities via data transfers to and from the shared memory. However, generally there are significant differences in the way the allocation of processors is handled by MIMD multiprocessor systems, and the way processor allocation is done in the present *FrameWorks* system. Processor allocation in the present system is directed towards the needs of the workstation environment. In a multiprogrammed multiprocessor system, for example, an application may not get scheduled for execution, if it requires a fixed number of processors and the given number of processors are not available. Under such circumstances, the *FrameWorks* model has the advantage that it allows applications to be easily restructured according to the available number of processors. Tightly coupled multiprocessors generally have much smaller interprocess communication costs as compared to the loosely coupled workstation environment. Therefore, another implication of implementing *FrameWorks* in a tightly coupled environment would be that it would be possible to use the model for exploiting much finer grains of concurrencies.

6.3. Potential Enhancements to the FrameWorks System

This section deals with possible directions for the evolution of the *FrameWorks*

system. Presently, the two versions of the *FrameWorks* systems have been implemented under the Sun OS 3.5 operating system environment equipped with Network File System (NFS). Due to certain limitations of the software environment, it was not possible to implement some desirable features without considerable effort. Also, some new components may be added to the system that would aid a user during various phases of developing a parallel application. Finally, in future, for developing a commercial product based on the *FrameWorks* model, it may be possible to increase the efficiency and flexibility of the system through the use of lower level development tools. Such possible enhancements to the system are discussed in this section.

6.3.1. Process Migration

In a workstation environment, most of the processors are owned by groups or individuals who have priority over other users in using these processors. Ideally, a shared processing facility in such an environment should utilize only idle processors. However, a difficult situation arises if, in the middle of an execution, the owner of a workstation that is being used by a parallel application, begins to use his workstation. Ideally, a system meant for shared processing should relinquish this processor from the application without aborting the entire application. Such relinquishing of a processor may be based either on the presence of a certain user on the workstation, the presence of another active task, the load average on the workstation, or a combination of such criteria. In *FrameWorks*, it is possible to release a workstation in the middle of an execution only if the desired workstation is being used as an *employee* of a *contractor*. As described in chapter 3, an *employee* processor belonging to a *contractor* can be released when the *employee* completes the unit of work currently assigned to it by the *contractor*. Since, *employee* processors generally constitute the major bulk of processors used by parallel applications, *FrameWorks* is partially able to fulfill its goal of not obstructing the owners of workstations. However, presently it is not possible to release an arbitrary processor at an arbitrary point of time during the execution of a parallel application. However, it may be possible to achieve this goal if the underlying

operating system provided a facility called as process migration. *Process migration* is the relocation of a process from the processor on which it is executing (the *source* processor) to another processor (the *destination* processor) in a distributed system [85]. It is desirable that the migration of the process should result in minimal interference with the execution of the application and the rest of the system. Also, a migrated process should not continue to depend on the *source* processor, once it is executing on the *destination* processor [111]. To the best of our knowledge, the facility of process migration is still not present in commercially available distributed environments. However, several research oriented distributed operating systems have proposed some kind of process migration facility [85,108,111]. The current implementation of *FrameWorks* has been done under Sun OS operating system which does not offer any process migration facility. Therefore, we were unable to experiment with releasing of an arbitrary processor in our current implementation. However, the development of *FrameWorks* under an operating system that offers process migration, such as V-System [23], would allow it to be a more flexible and attractive tool.

6.3.2. An Integrated User Interface

In *FrameWorks*, to develop parallel applications, a user is required to specify two types of information: the application graph that specifies the parallel structuring of the application, and source code for various modules of the application. The application graph is prepared using graphic interface provided by the *Module_Craft*, whereas the source code for the application may be prepared using a conventional text editor such as *vi* [15] or *emacs* [106]. A second graphic interface, called *View_Graph*, is used for post-execution analysis and tuning of the application. Other steps such as preprocessing, compilation, and linking of modules are carried out separately as described in chapter 5. A desirable extension of the *FrameWorks* system would be to have a single integrated high level graphic interface that not only includes *Module_Craft* and *View_Graph* but also incorporates subsystems for source code preparation, preprocessing, compilation, and linking at its lower levels.

6.3.3. Better Communication Protocols for LAN-Based Environment

FrameWorks system uses the Network Multiprocessor Package (NMP) [76] for process creation and inter-process communication. NMP, which is described in the next section, is built upon the socket abstraction of Unix [71]. It provides options of using either User Datagram Protocol (UDP) [84] or Transmission Control Protocol (TCP) [83] as the underlying protocol for message passing. Since, the UDP protocol does not guarantee reliable delivery of messages, the system using this protocol is responsible for providing mechanisms for handling lost, duplicate, or out of order messages. This makes implementation of a system using UDP much more complicated. The TCP protocol, on the other hand, provides reliable and ordered delivery of messages. However, the TCP protocol is less efficient than the UDP protocol. Originally, the TCP protocol was designed for bulk data transfers over wide area networks (WANs). In local area network (LAN) based environments, generally, transmission error rates are much lower, message sizes are smaller, and the frequency of transmission is higher as compared to the WANs. Several researchers have pointed out that TCP's algorithm for message delivery is not suited for interactive applications in a LAN-based environment [72,118]. Also, the TCP protocol has no special support for specific types of message passing protocols such as remote procedure calls, multicast messages, and broadcast messages. Some more recently proposed transport protocols attempt to overcome these shortcomings [25,30,89]. Future versions of *FrameWorks* should also consider the possibility of using a protocol which more specifically addresses the needs of LAN-based application environments. This may not only result in a more efficient implementation but may also simplify the implementation work itself.

6.3.4. Extension and Refinement of Templates

The present repertoire of templates in the *FrameWorks* system consists of three types of input templates, three types of output templates, and two types of body tem-

plates. However, a template-based system is not restricted to having a fixed number of templates. In fact, a system may evolve by refining its present set of templates, or by adding new templates to its collection. Ideally, a new or modified template should impose the least number of constraints on the use of the template in conjunction with other templates. Also, a new template should conform to the overall goals of the environment such as, handling of termination of application and load monitoring.

As an example, the functionality of the *executive* template in the system may be extended. Currently, only the *main* module of an application is allowed to use the *executive* type body template. The main reason for this constraint is that, in the current implementation, it is not possible to redirect standard input or standard output of a process to some other (possibly remote) process. Removing this restriction would imply that any module in the application may redirect its standard input or output to a remote terminal. Besides being useful, this type of extension is not likely to impose any new restriction on the use of this template.

6.3.5. Developing Support Subsystems

The set of components, presently available in the *FrameWorks* system, provide basic tools required for developing parallel applications. Chapter 5 also describes the design of *View_Graph* component which can be used for visualization, debugging, and tuning purposes. However, there are some other components, that may be required for a more complete development environment.

One area in which more tools may be useful is that of parallel debugging. In a user survey conducted on the USENET [112], users of parallel programming tools stressed the significance of visual tools for replaying the execution of a parallel application. Other popular debugging methods included use of print statements, and parallel symbolic debuggers. The *View_Graph* component of *FrameWorks* provides a visual tool that can be used for tuning and debugging of parallel applications. Also, it would not be difficult to integrate debugging methods into the *FrameWorks* environ-

ment that are based on printing execution traces into NFS files. Parallel versions of symbolic debuggers may also be useful, however, development of such debuggers may require significantly much more effort.

The above-mentioned survey also reveals that a common method of developing parallel applications is to initially debug the application using one processor. This is followed by debugging the application using minimum number of necessary processors. The application is finally debugged with the targeted number of processors. In *FrameWorks*, it is particularly easy to follow this development method. First, the close structural similarity between a sequential program and its parallel version allows an application to be initially developed sequentially. When the sequential version is ready, its parallel version can be developed with small amount of effort. Second, the method for allocating processors to modules allows the whole application to be run on any number of processors. Third, the number of replicas employed by a *manager* or *contractor* template can be adjusted without any change in the application code.

A tool for managing source and object code libraries is another potentially useful component that may be developed for the *FrameWorks* environment. In the sequential case, the *make* utility [41] of Unix is often used for this purpose. However, the task of managing libraries of code becomes more complicated in a heterogeneous environment for parallel processing. There is due to several factors. First, the hardware environment may contain processors of different types. Second, a single application consists of several executable modules. Third, generally a larger number of steps are involved in preparing executable modules for a parallel application. Specifically, in the context of the *FrameWorks* system, a code management subsystem would at least meet the following requirements:

- 1) Recompilation of modules of an application whose source code or associated ".h" files have changed, thus necessitating the recompilation of the modules.
- 2) Recompilation of modules whose *FrameWorks* generated code has changed due to changes in the application graph.

- 3) Recompilation of modules whose allocated processor type has changed since the last execution.

Other tools that may aid in designing, partitioning, or analyzing a parallel application may also be developed. The nature of tools that would be useful in the *FrameWorks* environment will gradually become clear as the model is used for developing more real life parallel applications [92].

6.4. Tools Used for Developing FrameWorks

The *FrameWorks* system has been developed under the Sun OS operating system environment which is extended by the Network File System (NFS). As described in Chapter 5, most of the tools used for developing the system, such as *lex*, *C* language compiler, and *RPC* protocol subsystem, are familiar software tools present in the Sun OS-based environment. In this section, however, we describe two special tools developed at the Department of Computing Science, University of Alberta, that significantly aided the development of the system.

The first of these tools is actually a collection of user interface management subsystems [101]. The major tools of this collection are called *Diction*, *Chisel*, and *Vu*. These tools allowed easy and quick generation of screen layouts for *Module_Craft* and *View_Graph*. With these tools, the user specifies the desired dialogue or command interface for his graphic interface. Based on this specification, the generated interface handles communication of input and output parameters between the interface and underlying application code. The application code is written by the user which deals specifically with the functionalities of the application such as, processing of input data, management of application's data structures, drawing of graphical icons, and checking of semantic constraints. These tools proved quite useful for quick generation of visual user interfaces.

The other locally developed tool used was the Network Multiprocessor Package (NMP) [76]. NMP is based on the socket abstraction of Unix [71] and provides a

higher level message passing interface than the one provided by the socket mechanism. Input to NMP consists of a configuration file which specifies the executable files whose execution needs to be initiated on various processors, and the structure of channels of communication among the processes. After initiation, processes communicate by using library procedures such as *SendNode* and *RecvNode*.

During the development of the *FrameWorks* system, we had to experiment with several different graphic interfaces and communication protocols. NMP and the user interface management subsystems provided us with tools which are appropriate for development work that is exploratory in nature. These tools provide significant savings in terms of development time. However, since both of these tools are not widely available, it affects the portability of the system. Also, the high-level nature of these tools takes away some of the flexibility and fine control that is generally available in low-level tools. For example, with NMP, it is not possible to establish a new communication channel at execution time between two arbitrary processes. In spite of such shortcomings, these high-level prototyping tools were of tremendous value during the development of the system. In a exploratory research work such as this, use of such tools allows rapid development of the system.

6.5. Experience With FrameWorks

The *FrameWorks* system has been used at the Department of Computing Science, University of Alberta by several persons for developing their parallel applications. So far, over half a dozen applications have been developed using the system. Our experience with the system indicates that in the case of pre-existing sequential applications, a fair amount of performance can easily be obtained through the simple modifications needed to parallelize the applications. In several cases, partitioning of the complete application into modules was possible while keeping most of the code in the sequential version intact. However, in some cases more efficient partitioning of modules did require moderate amounts of work. In the case of applications that were designed with

the *FrameWorks* system in mind, the amount of work required to switch between sequential and parallel versions was quite small. Also, restructuring the applications for experimenting with different templates often required either no modifications or only a small amount of modifications within the modules.

The network diagrams produced by the graphic interface *Module_Craft* seem rather unconventional. However, they make the parallel structure of the application explicit. The assistance that such an interface can provide in visualizing, documenting, and debugging the application makes it worth the effort. The information provided to *Module_Craft* by the user can later be used by *View_Graph* for post-execution replay and the display of important execution statistics associated with a particular execution of an application. Such information can provide significant aid in debugging an application, or tuning it for better performance without having to resort to low-level tracing of remote processes. Therefore, in retrospect, the decision of developing *FrameWorks* as a visual system appears quite natural, although in the beginning we did have some doubts.

6.6. Conclusions

Our experience in developing the *FrameWorks* model and system, as well as while using the system, has taught us several lessons. It has refuted some of our earlier beliefs and strengthened others. First, although the *FrameWorks* system is a suitable tool for developing parallel applications in a workstation environment, the scope of utility for the template-based approach is not restricted to the workstation environment. Different types of parallel development environments can be developed using the template-based approach depending upon the underlying hardware or software environment. The template based approach can also be used for developing customized tools that are suitable for developing particular classes of parallel applications. Second, the development of *FrameWorks* system has demonstrated the practical feasibility of the model. The patterns of usage that would emerge, when the system is used for

different types of real-life applications, would be quite important. These patterns should help determine further evolution of the system. And finally, the presence of a process migration facility and use of more efficient and flexible LAN-based transport protocols would result into a more efficient, flexible, and pleasant to use system for the workstation environment.

Chapter 7

Critical Review and Concluding Remarks

7.1. Introduction

The purpose of this chapter is two-fold: to compare the work described in this thesis with other relevant research works, and to present a summary of the thesis. The following section analyzes and compares the template-based model of *FrameWorks*, as well as the *FrameWorks* system from several viewpoints. Section 7.3 presents the thesis summary. Finally, we close this thesis with some concluding remarks in section 7.4.

7.2. Comparison with Relevant Research Works

FrameWorks can be analyzed from several viewpoints. First, it is a system aimed to provide structuring of parallel applications in a workstation environment. Second, templates can be looked upon as a prepackaged set of macros whose attachment to a module causes automatic generation of code. Third, *FrameWorks* provides a visual environment for development of parallel applications. There have been several relevant research efforts in all of these areas and development of *FrameWorks* was influenced significantly by them. In this section, we attempt to compare and analyze the present research work from these different perspectives.

7.2.1. *FrameWorks* as a System for Shared Processing

In the past, the idea of providing shared processing on a computer network has been studied by several researchers [23, 52, 63, 65, 80, 94]. Among the earlier efforts, the worm programs of Shoch and Hupp [94] developed some distributed applications in a workstation environment and demonstrated the feasibility of dynamically expanding and shrinking programs that utilized idle processors. However, their work did not concern itself with the development of any conceptual model or language for this purpose. Some other systems are based on managing a pool of available processors and

supply an application with an idle processor, or a processor with low load average on demand [52,63]. Yet another class of approaches are based on distributed operating systems [23,85,108,113]. Some systems allow the allocated processor to be reclaimed by a user who has a higher priority for using the processor [80]. *FrameWorks* not only keeps track of the available processors but also offers a conceptual model for developing parallel applications, which is consistent with the constraints of the workstation environment. Using this model, an application can be easily modified to run with the available number of processors. *FrameWorks* can be compared with the system by Jagannathan et al. [65] which provides a model for parallel programming using a network of workstation. The underlying model is based on the data flow methodology. Processors are assigned at compile time by the user to the nodes of the data flow graph. Unlike *FrameWorks*, there is no provision for using variable number of available processors in a computation.

In a survey of distributed computing systems, Bal and Tanenbaum [5] classify the approaches to mapping of processors to processes into three categories. According to these approaches, the mapping of processors can be fixed at compile time, fixed at execution time, or not fixed at all. In *FrameWorks*, processors can be assigned to modules before compilation. However, the actual assignment may change before execution depending upon the programmer's preferences and available processors. Also, the *contractor* template allows processors to be gracefully added or removed from a running application. However, we have not been able to experiment with releasing of an arbitrary processor from a running application for the lack of a process migration facility in the Sun OS operating system. Bal and Tanenbaum [5] also distinguish approaches to mapping of processors to processes depending on whether the mapping is performed under user's control or system's control. In *FrameWorks*, a programmer decides about the desired amount of control to be exercised in allocation and mapping of processors. One may move from one extreme, whereby a user exercises complete control, to the other extreme where the whole matter may be left to the discretion of

the system.

7.2.2. Templates and Related Macro-expansion Based Approaches

FrameWorks proposes a new approach to developing distributed applications in which parallel operations are encoded via templates. Each template can be looked upon as a collection of macros [98]. Attachment of a template to a module causes these macros to be inserted and expanded at appropriate places in the user's code to handle low level synchronization and communication. Macro-expansion based techniques have been used in the past: a technique based on macro-expansion has been used to create parallel Fortran programs [67]. Babb uses a technique based on macro expansion to restructure sequential Fortran programs to produce an acyclic network of processes that can be executed in parallel [64]. However, in both cases, the prescribed macros are at a much lower level and the programmer must insert them in the source code. In contrast, templates are attached externally to a module. Each template may represent encapsulation of several macros that are automatically inserted and expanded at appropriate places in the source code. This makes the use of templates much simpler and allows significant flexibility in restructuring the application. Also, different templates can be used with the same procedure in different applications.

There is a more significant difference between the above-mentioned macro-expansion based approaches and the templates of *FrameWorks*. Each template represents a conceptual chunk of some frequently used parallel configuration and its associated operations. Therefore, unlike expansion of a usual macro, templates generate code which is non-contiguous and is also sensitive to adjacent configurations. Such grouping of low-level pieces of information into high-level chunks is in accordance with the chunking theory of cognitive psychology [96], which suggests that humans often abstract low-level details into high-level conceptual chunks for frequent and efficient use.

7.2.3. FrameWorks as a Visual System

FrameWorks implements a high-level visual system for distributed computing in a workstation environment. The idea of visual systems for parallel programming is not new. In the past, several researchers have proposed graphical languages based on specific kinds of graphs such data flow graphs [32,69], or hierarchical graphs [103]. However, until recently, there was a paucity of actual implementations. The poker programming environment [104] was a pioneering effort in visual programming that allowed grid-based specification of a graph showing communication links among parallel processes in a program. Tomboulia et al. [107] implemented a visual programming environment for programming functional units on a single node of a Navier-Stokes computer. In late eighties, several other important visual systems for parallel computing were developed [16,17,50,73]. Nowadays, it has almost become fashionable to have a visual interface for a tool or system. However, the important issues in this context are the suitability of a visual interface for the underlying representation, the types of inputs required from the user for generating application specific visual interfaces, the aspects of parallel application development that can be captured by the visual interface, and the degree of details presented by the interface. Some of the above systems only support program visualization [16,17], whereas some others only support execution visualization and debugging [4]. Only a few systems support both activities [50,73]. In some visual systems, a programmer is required to program his application specific visual interface in the notation provided by the system [4]. Also, it should be noted that while visual interfaces may enhance the look or presentation of a system for parallel programming, their incorrect design may have very unpleasant side-effects. For example, if a visual interface is tied to the actual run-time component of the system, the slow speed graphic terminals or workstations may become a bottleneck while displaying the behavior of a program. Also, the necessary synchronization of the visual component with the rest of the system may introduce time-dependent bugs in the program.

In these respects, the graph-based representation of *FrameWorks* is inherently suitable for development as a visual system. The interface can support visual program development, execution animation, and performance debugging. The same application graph as specified by the user can be used by the visual interface of *Module_Craft* as well as *View_Graph*. For the reasons mentioned above, none of the interfaces are directly coupled with the actual run-time environment of the system. However, we feel that currently, the visual interface of *View_Graph* lacks sophistication, in terms of its capability to display details of various data structures, in comparison to some other systems [50,73].

7.2.4. Comparison with Other Languages, Tools, and Systems

Comparison of *FrameWorks* with general purpose concurrent languages [8,53,53,57], or low-level distributed computing tools [9,71], reveals some interesting benefits as well as some limitations of our template-based approach.

- 1) Unlike CSP or Ada, *FrameWorks* does not embody a complete language for parallel programming. Rather, it proposes a new parallel programming model which can be implemented on top of an existing sequential language like Pascal or C.
- 2) There is a large class of distributed applications where synchronization, communication, and scheduling structures are much more restricted. It has been pointed out that such distributed applications can be generated more easily via specialized languages/systems than by general purpose concurrent languages [18,78]. *FrameWorks*, along with several other systems, can be put in the category of specialized systems/languages that aim at providing easy generation of commonly occurring forms of synchronization and concurrency [18,64,67,78]. For large, general-purpose programs such as an operating system or a real time transaction system where a group of processes interact or "converse" in an arbitrary manner [44], *FrameWorks* may not be a suitable tool. General purpose concurrent

languages such as CSP or Ada, or low-level tools such as sockets, may be more appropriate for such situations.

- 3) Although *FrameWorks* provides a easy way of constructing distributed applications, it hides overhead and communication costs from the user. Consequently, the final design may not be as efficient as one that is custom built by an expert programmer. Customized generation of programs using low-level tools, such as sockets, is a time consuming and tedious process even for experienced programmers. However, it might be a suitable alternative for applications that critically depend on achieving maximum efficiency.
- 4) Most high-level concurrent languages do not allow a programmer to map his/her processes to processors. This is a handicap, especially in a workstation environment due to the nature of division of ownership rights commonly found in such environments. *FrameWorks* provides a high-level notation in which a programmer can express his/her processor allocation constraints if desired.
- 5) Although the contractor/employee combination involves dynamic scheduling and termination of processes, for simplicity *FrameWorks* does not support dynamic initiation of processes by the user. Therefore, it is more suitable for environments where the computational structure of an application can be specified at compile time. For more complex situations where the pattern of computation is unknown at compile time, languages such as Ada or other low-level tools [23,71] might be more appropriate.

In summary, *FrameWorks* is a special purpose tool for the generation of applications that can be structured in terms of commonly occurring communication/synchronization patterns. Such applications can be easily, quickly and correctly generated using *FrameWorks* and they take advantage of available workstations in a flexible manner. However, the approach may not be suitable for programs with complicated communication or synchronization structures or programs that aim to attain very high levels of efficiency.

7.3. Thesis Summary

Chapter 1 introduces the issue of shared processing in a workstation environment and describes the involved difficulties. The scope of the thesis is defined and intended course for the rest of the thesis is outlined.

Chapter 2 presents a review of the basic interaction techniques for parallel computing. The state of the research in systems, tools, and languages, directed towards parallel computing using a network of computers (especially workstation LANs), is described.

Chapter 3 gives an informal description of the *FrameWorks* model. The concept of *template-attachment* is introduced. Various templates of the model are described. The function of the model is illustrated through a detailed example.

A formal set-theoretic description of the structure of a *FrameWorks* application is given in Chapter 4. Also, a formal description of various templates of the model is given using a technique based on the *General Transition Model*. Based on these formal descriptions, two implementation models are developed and their communication protocols are specified. These implementation models bring out the significant issues involved in implementing the *FrameWorks* model.

Chapter 5 deals with implementation of the *FrameWorks* model. Various components of the two versions of the *FrameWorks* systems, based on the two implementation models of Chapter 4, are described. Several types of experiments to examine the performance levels that can be achieved with the two implemented versions, are described. Results of these experiments are presented.

There are a number of possible directions in which the work described in this thesis can be extended. These extensions are outlined in Chapter 6. The description brings out some aspects and applications of the template-based approach that are generally beyond the scope of the thesis. The discussion in the chapter also highlights the areas in which the *FrameWorks* system can be extended.

Chapter 7 compares the work described in this thesis with other relevant research works from several different perspectives. It presents a summary of the thesis and its conclusions.

7.4. Conclusions

Parallel computing using a network of workstations is an idea which is continually being rediscovered by researchers in several different forms. The thesis proposes the *FrameWorks* model for developing and restructuring parallel applications in the workstation environment. The graph-based representation of the model naturally lends itself to be developed as a visual system. Practical feasibility of the model is demonstrated by developing two versions of the system based on the model. Experimental results indicate the viability of coarse grain parallel computations using workstation LANs. A specific template of the system, called *contractor*, provides a new method for distributing work in an environment with varying numbers of available processors. As discussed in Chapter 6, the concept of *contractors* can be expanded to structure applications with expanding and shrinking hierarchies of computations. Although, *FrameWorks* is expected to make it easy to develop, as well as to restructure applications to match them to available resources, it is too early to comment on the effectiveness and convenience of the system for the users with real applications. The programming and usage styles that would develop when the system is widely used would be particularly important. The resulting feedback would have major influence on the further evolution of the system. The *FrameWorks* model itself is based on the concept of *template-attachment*, a new concept which is introduced in this thesis. It seems that the template-based approach is much more versatile and can be employed under different hardware and software environments for developing parallel computing tools with specific system goals and flavors. The *FrameWorks* system is the first implementation based on this approach, and hopefully, it will not be the last.

References

1. G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, pp. 3-43, March 1983.
2. G. R. Andrews and R. A. Olsson et al., "An Overview of the SR Language and Implementation," *ACM Trans. Prog. Lang. Syst.*, vol. 10, no. 1, pp. 51-86, Jan. 1988.
3. W. C. Athas and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *Computer*, vol. 21, no. 8, pp. 9-24, Aug. 1988.
4. M. L. Bailey, D. Socha, and D. Notkin, "Debugging Parallel Programs Using Graphical Views," *Proc. International Conference on Parallel Processing*, vol. II, pp. 46-49, 1988.
5. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing System," *ACM Computing surveys*, vol. 21, no. 3, pp. 261-322, September 1989.
6. D. H. Ballard and C. M. Brown, "Scene Labeling and Constraint Relaxation," in *Computer Vision, Section 12.4*, pp. 408-430, Prentice-Hall Inc., 1982.
7. R. M. Balzer, "PORTS - A Method for Dynamic Interprogram Communication and Job Control," *Proc. AFIPS Joint Spring Computer Conf.*, vol. 38, pp. 485-489, May 1971.
8. J. G. P. Barnes, "An Overview of Ada," *Software Practice and Experience*, vol. 10, pp. 851-887, 1980.
9. A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, pp. 38-59, Feb. 1984.
10. A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in Emerald System," *Proc. of Object Oriented Programming Systems, Languages, and Applications*, 1986.

11. A. P. Black, "Supporting Distributed Applications: Experience with Eden," *Proc. of 10th ACM Symp. on Operating Systems Principles*, pp. 181-193, 1985.
12. G. Bochmann, *Concepts for Distributed Systems Design*, pp. 139-209, Springer-Verlag, 1983.
13. G. Bochmann, "A General Transition Model for Protocols and Communication Services," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 643-650, April 1980.
14. K. S. Booth, J. Schaeffer, and W. M. Gentleman, "Anthropomorphic Programming," *Tech. Rep. CS-82-47*, Dept. of Computer Science, University of Waterloo, Feb., 84.
15. S. R. Bourne, *The UNIX System*, Addison-Wesley Publishing Company, 1983 .
16. G. P. Brown and S. M. Sobek et al., "Program Visualization: Graphical Support for Software Development," *Computer*, vol. 18, no. 8, pp. 27-35, August, 1985.
17. J. C. Browne, M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, pp. 10-18, July, 1989.
18. G. Butler and M. J. Kendall, "The Suitability of Master/Slave Concurrency of Concurrent Euclid, Ada, Modula," *Software-Practice and Experience*, vol. 17, no. 2, pp. 117-134, Feb. 1987.
19. R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science*, vol. 16, pp. 89-102, Springer-Verlag, New York, 1974.
20. N. Carriero and D. Gelernter, "Linda in Context," *CACM*, vol. 32, no. 4, pp. 444-458, April 1989.
21. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw Hill Book Company, 1984.
22. K. M. Chandy, L. M. Haas, and J. Mishra, "Distributed Deadlock Detection," *ACM Transaction on Computer Systems*, vol. 1, no. 2, 1983.

23. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations," *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp. 129-140, Dec. 1983.
24. D. R. Cheriton, "Multi-Process Structuring and the Thoth Operating System," *Ph. D. Dissertation*, Department of Computer Science, University of Waterloo, 1978.
25. D. R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of the Communication Systems," *Proc. ACM SIGCOMM*, pp. 406-415, Aug. 1986.
26. M. F. Cohen and D. P. Greenberg, "The Hemi-Cube: The Radiosity Solution for Complex Environments," *Proc. of ACM SIGGRAPH*, pp. 31-40, 1985.
27. S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination," *Proc. Symposium on Principles of Distributed Computing*, pp. 29-33, 1982.
28. M. E. Conway, "Design of a Separate Transition Diagram Compiler," *CACM*, vol. 6, no. 7, pp. 396-408, July 1963.
29. M. E. Conway, "A Multiprocessor System Design," *Proc. AFIPS Fall Joint Computer Conference*, pp. 139-146, Nov. 1963.
30. R. P. Cook, G. Chesson, B. Eich, V. Chryver, A. Cherenson, and A. Whaley, "XTP Protocol Definition Revision 3.4," *Protocol Engines Inc.*, vol. SE-6, no. 6, pp. 563-571, 1988.
31. I. M. Copi, *Introduction to Logic*, MacMillan Publishing Co., Inc., New York, 1982.
32. A. L. Davis and S. A. Lowder, "A Sample Management Program in a Graphical Data Driven Programming Language," *Compcon Spring 81*, pp. 162-167, February, 1981.
33. H. M. Deital, *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Revised First Edition, 1984.

34. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *CACM*, vol. 9, no. 3, Mar. 1966.
35. A. K. Dewdney, "Computer Recreations," *Scientific American*, pp. 16-26, August, 1985.
36. E. W. Dijkstra, "Cooperative Sequential Processes," in *F. Genuys (Editor), Programming Languages*, Academic Press, New York, 1968.
37. E. W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *CACM*, vol. 18, no. 8, pp. 453-457, Aug. 1975.
38. E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Operating Systems Techniques*, R. H. Perrot (ed.), pp. 72-93, Academic Press, New York, 1972.
39. E. W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982.
40. J. A. Feldman, "High Level Programming for Distributed Computing," *CACM*, vol. 22, no. 6, pp. 353-368, June 1979.
41. S. I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, April, 1979.
42. N. Francez, "Distributed Termination," *ACM TOPLAS*, vol. 2, no. 1, 1980.
43. J. Gaschnig, "A Constraint Satisfaction Method for Inference Making," *Proceedings of the 12th Allerton Conference on Circuit and System Theory*, Urbana, Illinois, October 1974.
44. E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*, pp. 225-270, Digital Press, 1977.
45. D. Gelernter and A. J. Bernstein, "Distributed Communication Via Global Buffer," *Proc. Symposium on Principles of Distributed Computing*, pp. 10-18, 1982.
46. M. L. Gick and K. L. Holyoak, "Schema Induction and Analogical Transfer,"

- Cognitive Psychology*, vol. 15, pp. 1-38, 1983.
47. C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces," *Proc. of ACM SIGGRAPH*, pp. 213-222, 1984.
 48. J. N. Gray, "Notes on Database Operating Systems," *Operating Systems an Advanced Course*, Springer Verlag, 1978.
 49. M. Green and J. Schaeffer, "FrameWorks: A Distributed Computer Animation System," *Proc. of Canadian Information Processing Society Edmonton '87*, pp. 305-310, 1987.
 50. V. A. Guarna and D. Gannon et al., "Faust: An Integrated Environment for Parallel Programming," *IEEE Software*, vol. 6, no. 4, pp. 20-27, July, 1989.
 51. A. N. Habermann, "Prevention of System Deadlocks," *CACM*, vol. 12, no. 7, pp. 373-377, 1969.
 52. R. Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," *The 6th Intl. Conf. on Distributed Computing Systems*, pp. 260-267, May 1986.
 53. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *CACM*, vol. 21, no. 11, pp. 934-941, Nov. 1978.
 54. P. Brinch Hansen, *Operating Systems Principles*, Prentice-Hall, 1973.
 55. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp. 199-206, June 1975.
 56. P. Brinch Hansen, "Structured Multiprogramming," *CACM*, vol. 15, no. 7, pp. 574-578, July 1972.
 57. C. A. R. Hoare, "Communicating Sequential Processes," *CACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.
 58. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *CACM*,

- vol. 17, no. 10, pp. 549-557, Oct. 1974.
59. C. A. R. Hoare, "Towards a Theory of Parallel Programming," in C. A. R. Hoare and R. H. Perrott (Editors), *Operating Systems Techniques*, Academic Press, New York, 1972.
 60. R. C. Holt, "Comments on Prevention of System Deadlocks," *CACM*, vol. 14, no. 1, pp. 36-38, Jan. 1971.
 61. R. C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179-196, Sept. 1972.
 62. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill Book Company, 1984.
 63. K. Hwang, W. J. Croft, and G. H. Goble et al., "A Unix-Based Local Computer Network with Load Balancing," *IEEE Computer*, pp. 55-65, April 1982.
 64. R. G. Babb II, "Parallel Processing with Large Grain Data Flow Techniques," *Computer*, vol. 17, no. 7, pp. 55-61, 1984.
 65. R. Jagannathan, A. R. Downing, W. T. Zaumen, and R. K. S. Lee, "Dataflow-based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations," *Proc. International Conference on Parallel Processing*, pp. 209-216, 1989.
 66. A. Jones and P. Schwarz, "Experience Using Multiprocessor System - A Status Report," *Computing Surveys*, vol. 12, no. 3, pp. 121-166, June 1980.
 67. H. F. Jordan, "Structuring Parallel Algorithms in a MIMD, Shared Memory Environment," *Parallel Computing*, vol. 3, no. 2, pp. 93-110, May 1986.
 68. E. G. Coffman Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67-78, 1971.
 69. R. M. Keller and W-C. J. Yen, "A Graphical Approach to Software Development Using Function Graphs," *Proc. COMPCON Spring*, CS Press, pp. 156-161, California, 1981.

70. A. M. Kobos, R. E. VanKooten, and M. A. Walker, "The Myrias Parallel Computer System," in *Algorithms and Applications on Vector and Parallel Computers*, pp. 103-117, Elsevier Science Publishers, 1987.
71. S. J. Leffler, W. N. Joy, and R. S. Fabry, "4.2BSD Networking Implementation Notes," *University of California at Berkley*, July 1983.
72. S. J. Leffler, M. KKMckusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, 1989 .
73. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman, "Visualizing Performance Debugging," *Computer*, October, 1989.
74. B. Liskov, "Primitives for Distributed Computing," *Operating Systems Review*, vol. 13, no. 5, pp. 33-42, Dec. 1979.
75. M. Maekawa, O. E. Oldehoeft, and R. R. Oldehoeft, *Operating Systems: Advanced Concepts*, The Benjamins/Cummings Publishing Company, Inc., 1987.
76. T. A. Marsland, T. Breitzkreutz, and S. Sutphen, "A Network Multi-Processor for Experiments in Parallelism," *Concurrency: Practice and Experience*, vol. 3, no. 3, pp. 203-219, June 1991.
77. J. T. McCall, J. G. Tront, F. G. Gray, R. M. Haralick, and W. M. McCormack, "Parallel Computer Architectures and Problem Solving Strategies for Consistent Labeling Problem," *IEEE Transaction on Computers*, vol. C-34, no. 11, pp. 973-980, November 1985.
78. P. Mehrotra and T. W. Pratt, "Language Concepts for Distributed Processing of Large Arrays," *Proc. Symposium on Principles of Distributed Computing*, pp. 19-28, Aug. 1982.
79. D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. on Software Engg.*, vol. SE-5, no. 3, 1979.

80. D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Proc. Eleventh ACM Symp. On Operating Systems Principles*, pp. 5-12, 1987.
81. M. Olafsson and T. A. Marsland, "A Unix Based Virtual Tree Machine," *CIPS Congress 85*, pp. 176-181, June 1985.
82. G. Polya, *How to Solve It: A New Aspect of Mathematical Method*, Princeton University Press, Princeton, Second Edition, 1957.
83. J. Postel, "Transmission Control Protocol-DARPA Internet Program Protocol Specification," *RFC 793, Information Sciences Institute*, Sept. 1981.
84. J. Postel, "User Datagram Protocol," *RFC 768, Information Sciences Institute*, Aug. 1980.
85. M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. of Ninth ACM Symposium on Operating Systems Principles*, pp. 110-119, Dec. 1983.
86. M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill Book Company, 1987.
87. M. J. Quinn and N. Deo, "Parallel Graph Algorithms," *ACM Computing Surveys*, vol. 16, no. 3, pp. 319-348, Sept. 1984.
88. R. F. Rashid, "Threads of a New System," *Unix Review*, pp. 37-49, August 1986.
89. R. Van Renesse and A. S. Tanenbaum, "Performance of the World's Fastest Operating System," *ACM Operating System Review*, vol. 22, no. 4, pp. 25-34, Oct. 1988.
90. D. M. Ritchie and K. Thompson, "The Unix Time Sharing System," *The Bell system Technical Journal*, vol. 57, no. 6, Part 2, July-August, 1978.
91. J. Schaeffer, "Distributed Game-Tree Searching," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 90-114, 1989.
92. S.M. Shatz and J. Wang, "Introduction to Distributed Software Engineering,"

- Computer*, pp. 23-31, October 1987.
93. B. Shneiderman, *Designing the User Interface*, Addison-Wesley Publishing Company, 1987.
 94. J. F. Shoch and J. A. Hupp, "The Worm Programs- Early Experience with a Distributed Computation," *CACM*, vol. 25, no. 3, pp. 172-180, Mar. 1982.
 95. G. C. Shoja and R. G. Gurr, "Parallel Processing in Local Networks," *Proc. International Conference on Parallel Processing*, pp. 1048-1051, 1987.
 96. H. A. Simon and W. G. Chase, "Skill in Chess," *American Scientist*, vol. 61, pp. 394-403, July-August 1973.
 97. A. Singh, J. Schaeffer, and M. Green, "Graph-Directed Distributed Computing Using a Network of Workstations," *in preparation*, 1991.
 98. A. Singh, J. Schaeffer, and M. Green, "Structuring Distributed Algorithms in a Workstation Environment: The FrameWorks Approach," *International Conference on Parallel Processing*, vol. Vol. 2, pp. 89-97, August 1989.
 99. A. Singh, J. Schaeffer, and M. Green, "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstation," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 1, pp. 52-67, January 1991.
 100. A. Singh, J. Schaeffer, and M. Green, "A Template-Based Tool for Building Applications in a Multicomputer Network Environment," *Parallel Computing 89*, Leiden, The Netherlands, August, 1989.
 101. G. Singh and M. Green, "A High-Level User Interface Management System," *ACM SIGCHI'89*, New York, April 1989.
 102. G. Singh and M. Green, "Visual Programming of Graphical User Interfaces," *Workshop on Visual Languages*, pp. 161-173, Linkoping, Sweden, Aug. 1987.
 103. P. D. Slotts, "The PFG Language: Visual Programming for Concurrent Computation," *Proc. International Conference on Parallel Processing, Vol. II*, pp. 72-80,

1988.

104. L. Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, pp. 27-36, July, 1984.
105. M. H. Solomon and R. A. Finkel, "The Roscoe Distributed Operating System," *Proc. of Seventh ACM Symp. on Operating Systems Principles*, pp. 108-114, Dec. 1979.
106. R. Stallman, *GNU Emacs Manual*, Fifth Edition, Emacs version 18 for Unix Users, Oct. 1986.
107. S. Tambouliau, T. W. Crockett, and D. Middleton, "A Visual Programming Environment for the Navier-Stokes Computer," *Proc. International Conference on Parallel Processing, Vol. II*, pp. 67-71, 1988.
108. A. S. Tanenbaum and S. J. Mullender (Editors), *The Amoeba Distributed Operating System*, Vrije Universiteit, Amsterdam, The Netherlands, 1990.
109. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.
110. R. Taylor, "A. General Purpose Algorithm for Analyzing Concurrent Programs," *CACM*, pp. 362-376, May 1983.
111. M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V System," *Proc. Tenth ACM Symp. on Operating Systems Principles*, pp. 2-12, Dec. 1985.
112. P. S. Utter, "Parallel Debugging Survey Summary," *The Moderated Comp.parallel News Group of USENET*, May 1989.
113. B. Walker, G. Popek, and E. English et al., "The LOCUS Distributed Operating System," *Proc. of the Ninth ACM Symp. on Operating Systems Principles*, pp. 49-70, Dec. 1983.
114. P. Wegner and S. A. Smolka, "Processes, Tasks and Monitors: A Study of Concurrent Programming Primitives," *IEEE Trans. on Software Engineering*, vol.

- SE-9, no. 4, pp. 446-462, July 1983.
115. J. E. White, "A High Level Framework for Network Based Resource Sharing," *Proc. of National Computer Conference*, June 1976.
116. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software Practice and Experience*, vol. 7, pp. 3-35, 1977.
117. Y. Wong, "Efficient Methods for Radiosity Solution," *Ph.D. Thesis*, Department of Computing Science, University of Alberta, 1990 (To appear).
118. L. Zhang, "Why TCP Timers Don't Work Well," *Proc. ACM SIGCOMM Conference*, pp. 397-405, 1986.

Appendix A

The Overall Structure of the Distributed Radiosity System

```
-----
/* File containing definition of frames used by modules */

#define NUM_POLYGONS 400          /* number of polygons in each patch */
struct packet {
    int ply;
};

struct ff_matrix {
    int poly_no;
    int num_poly;
    float formfactor[NUM_POLYGONS];
};

-----
/* Structure of input module */

main()
{
    int i, start, interval;
    /* send out computations for specified patches */
    get_paras( );                /* get input parameters */
    for(i = start; i <= start + interval; i++){ /* loop through each patch */
        seq.ply = i;             /* select patch number */
        call_formfactor(seq);    /* call formfactor */
    }
}

-----
/* Structure of formfactor module */

#include <wintool.h>    /* include necessary definition files */
#include <UTILITY.h>

/* other global data structures declared here */

formfactor(seq)
struct packet seq;
{
    struct ff_matrix data_frame;
    read_image(seq);          /* read data related to the image */
    read_computation_data(seq); /* read set up parameters */
    compute_ff(seq);          /* compute form-factor matrix */
    call_organize( data_frame); /* send it to organize */
}

-----
/* structure of organize module */

organize(data_frame)
struct ff_matrix data_frame;
{
    FILE out_file;
    organize_data( data_frame);
    write_ff_matrix(out_file, data_frame); /* write the formfactor matrix on disk file */
}
-----
```

Appendix B

Some Example Applications

In this appendix, a few simple applications developed using the *FrameWorks* system are presented. *FrameWorks* has, so far, been used to develop over half a dozen applications. However, for brevity, not all the applications will be presented here. The applications have been developed by several different programmers at the Department of Computing Science, University of Alberta. The hardware computing environment consists of different models of Sun3 workstations which are connected by an Ethernet-based local area network. These examples are not intended to cover all the features of *FrameWorks*, but illustrate basic ideas of the system. Also, the implementations described here may not be based on the best available algorithm for the given problem.

B.1 The Prime Number Generator

The application *prime* implements a parallel program for searching all the prime numbers that lie between two given positive integer numbers. The complete application is divided into three modules. The application graph for *prime*, as prepared by the programmer using *Module_Craft*, is shown in Figure B.1. The module *Partition* receives the input parameters from the user which specify the interval to be searched, and number of subranges into which the whole search should be partitioned. The module also checks the validity of input parameters, prepares the required subranges to be searched, and creates the format for printing the results. All the information required to search a specific subrange is organized in a *frame* of *FrameWorks* and sent out to the module *FindPrimes* via a non-blocking call. As shown in Figure B.1, *Partition* is assigned a *manager* type output template. Thus, each call is processed by a single employee of *Partition*. Finally, the searched prime numbers are written into a file by *FindPrimes*.

In one experiment, prime numbers between 0 and 1,000,000 were computed. The whole range was divided into 20 subranges. The sequential execution of the application required 1057 seconds. In the parallel computation, a total of 10 workstations were used. Ten employees of *Partition* constituted ten replicas of *FindPrimes* that executed in parallel. Each replica of *FindPrimes* was assigned to a separate processor. The configuration yielded a parallel computation time of 118 seconds, thus providing a speed up of about 8.9 fold.

B.2 Mandelbrot Sets

Mandelbrot sets represent a fascinating application of fractal geometry, a mathematical study of forms having a fractional dimension [35]. They are sets of numbers computed in a complex plane. When plotted, these sets create attractive patterns with an extraordinary regularity. The computation of these sets, however, is a computation intensive task in which the value of each point on a selected square region of the complex plane is computed according to the given algorithm [35].

The parallel computation of Mandelbrot sets is based on the fact that each of the points in the selected region can be examined independently. However, assigning each single point to a separate processor would be an example of fine-grained parallelism that is not advantageous in a workstation LAN environment. In our example, to exploit the coarse grain concurrency in the task, the whole region is divided into rectangular subregions. The number of such subregions is determined by the programmer depending upon the number of available processors.

In our implementation, the complete application is divided into three modules (Figure B.2). The module *mandel* accepts the parameters from the user that specify the complex region to be explored, number of points to be assigned to the whole region, depth of iteration for each computation, and number of subregions into which the whole computation is to be divided. The size and location of each subregion along with the rest of the input parameters are passed to the *power_work* module via a non-

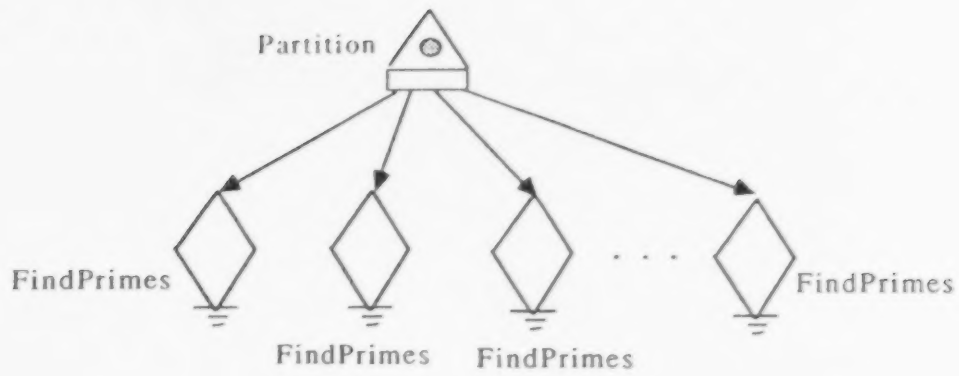


Figure B.1. Application Graph for Prime

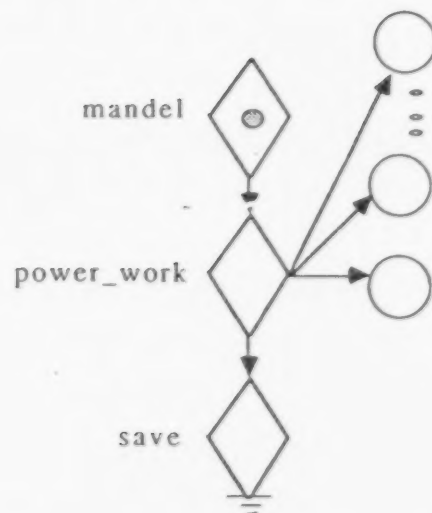


Figure B.2. Application Graph for Mandelbrot

blocking call. The *power_work* module computes the final value for each point in the specified subregion. The computed set of values are passed to the third module, called *save*, via a nonblocking call. *Save* writes the computed values in a stored file. All three modules work asynchronously. However, most of the computation is done by the *power_work* module. It is assigned a *contractor* template so that many replicas of *power_work* can operate in parallel.

In our experiment, a square region, with origin (-2, -1.25) and 2.50 as the length of each side, was investigated. The whole region was divided into $100 \times 100 = 10,000$ points. The depth of iteration for computing the value of each point was set at 1,000. For parallel computation, the complete square was divided into 18 rectangular subregions. The sequential computation took 424 seconds. In the parallel case, 6 workstations were used for executing 6 employees of *power_work* as well as *mandel* and *save*. The parallel computation took 99 seconds. This implies a speed up of 4.3 with 6 processors.

B.3 The Parallel Make System

The parallel make application, *Pmake*, is concerned with the parallel compilation of source files, and the linking of object files to create an executable program. It is based on the *Make* utility [41] of Unix. A file called *Makefile* is used to describe the structure of the application and the data dependencies between various source, object, and executable files. Each rule in the *Makefile* describes the name of the resultant file, a list of file(s) on which the resultant file(s) depend, and a list of commands that should be executed to produce the resultant file(s). A rule is activated if and only if any file on the list of its dependencies has been modified since its resultant file was last updated or created.

The computation graph for *Pmake* is shown in Figure B.3. The module *scheduler* examines the *Makefile* and creates a list of compilation and linking steps that need to be executed in order to complete the task. Each of the compilation steps is sent to the

module *compile* via a non-blocking call. The module *compile* performs the given compilation and informs the module *synch* about the completion of the step through a non-blocking call. When all the compilations are done, the module *synch* notifies *scheduler* about it. The *scheduler* then executes the final linking step to generate the desired executable file. Since, the compilation steps can be performed in parallel, the *compile* module is assigned a *contractor* template. Each replica of *compile* as well as *scheduler* and *synch* modules work in parallel.

In one of our experiments with the *Pmake* application, an old version of *Module_Craft* was generated. The sequential compilation and linking took 288 seconds. In the parallel version, 3 workstations were used. Three employees of *compile* were used as well. The parallel execution took a total of 132 seconds. Thus an overall speed up of 2.2 with 3 processors was obtained. The computation times include 46 seconds taken by the linking step which is basically done sequentially even in the parallel case. Excluding this time, the time taken by the compilation steps becomes 242 seconds in the sequential case and 86 seconds in the parallel case. This implies a speed up of 2.8 for the compilation steps alone.

B.4 The Animation System

This section describes an application for distributed animation that has been implemented using *FrameWorks* [49]. Computer animation involves creating a sequence of graphical images that, when shown in rapid succession, create the illusion of motion. The calculations required to produce these images are compute intensive and require a high volume of data. In the case of video, 30 images per second are required, each having a resolution of at least 512 x 512 pixels and each pixel representing 8 to 24 bits of information. Determining the value of a pixel may require hundreds of floating point operations.

The complete system consists of about 2500 lines code in our extended C language and is divided into three modules: *Model*, *PolyConv*, and *Split*. Figure B.4

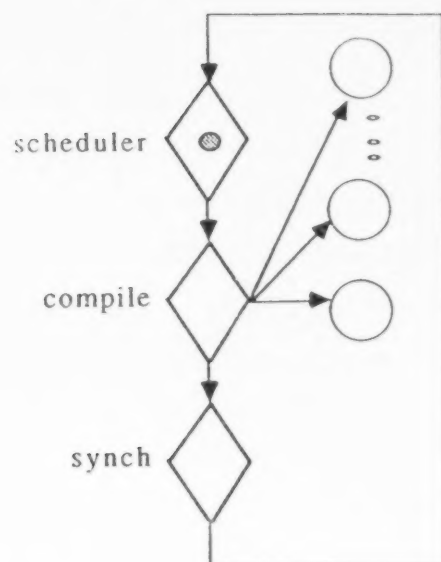


Figure B.3. Application Graph for Pmake

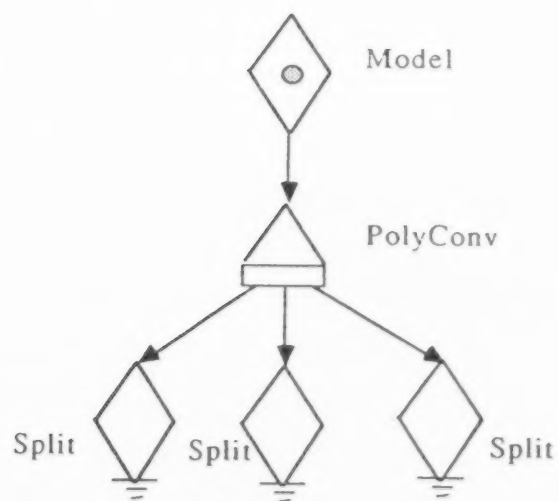


Figure B.4. Application Graph for Animation

shows one possible interconnection of these modules. The code in *Model* depends on the subject(s) of animation. In our application, *Model* deals with animating the simple schooling behavior of fish in a tank. The modeling and motion is computed through a model that takes into account behavioral characteristics of the fish and the laws of physics. *Model* computes the location and motion of each fish in the image and stores the results in a file on the distributed file system. Completion of this computation is communicated to *PolyConv* by issuing a non-blocking call. After this, *Model* continues with the computation of the next image in the animation sequence. At the same time, *PolyConv* reads the data from the disk and performs some data format transformations followed by viewing transformation, projections, sorting, and backface removal. *PolyConv* passes the data in the image along with the image's sequence number to *Split* module through a non-blocking call. *PolyConv* then waits to serve the next incoming call. The *Split* process performs hidden surface removal (using the z-buffer algorithm) and anti-aliasing. Finally the rendered image is stored on a disk file.

This application is an example where parallelism is exploited via partitioning as well as replication. However, the computation time requirements of *Split* module are much higher than those of *Model* or *PolyConv*. Therefore, *PolyConv* has been assigned a *manager* output template. Now, several replicas of *Split* work in parallel.

In one experiment with the application, 20 images of the above-mentioned animation-sequence were computed. The sequential computation took 1110 seconds. In the parallel version, 4 workstations were used. One workstation was assigned to the *Model* module. The rest three workstations were allocated to *PolyConv* and *Split* together. Three replicas of *Split* were used. With this mapping of processors, a parallel computation time of 334 seconds was obtained. This means a speed up of 3.3 with 4 processors.

B.5 Some Observations

Our simple examples, in this appendix, serve to illustrate several important points

about *FrameWorks*. First, these examples show that worthwhile performance gains are possible using the *FrameWorks* system with real life applications. Second, all the applications can be modified to run in a sequential environment through minor modifications. Third, as in the case of distributed radiosity application, several different configurations of each of the application described here can also be generated without any change in their application code. Finally, the *prime* as well as the *distributed animation* application were easily generated from already existing sequential versions of these applications.